FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Analyse von Lambda-Ausdrücken in Java

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Benjamin Fraller, BSc
Matrikelnummer 1027982

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam

Wien, 24. August 2018

Benjamin Fraller	Franz Puntigam

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Analyse von Lambda-Ausdrücken in Java

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Benjamin Fraller, BSc
Registration Number 1027982

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam

Vienna, 24th August, 2018

Benjamin Fraller	Franz Puntigam

Erklärung zur Verfassung der Arbeit

Benjamin Fraller, BSc
Pohlgasse 16/2/14

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. August 2018

Benjamin Fraller

Kurzfassung

Funktionale Konzepte sind wegen paralleler Programmierung und günstiger Mehrkernprozessoren sehr beliebt. Immer mehr objektorientierte Sprachen verwenden daher funktionale Konzepte. So wurden in Java 8 unter anderem Lambda-Ausdrücke und Streams hinzugefügt. Verhalten kann dabei in Form von Daten als Funktionen höherer Ordnung übergeben werden. Streams ermöglichen deklarativ eine sequenzielle und parallele Verarbeitung von Daten. In dieser Arbeit sollen Vor- und Nachteile der Konzepte und die Umsetzung, über Dekompilierung in JVM-Bytecode analysiert und mit anderen Sprachen verglichen werden. Die Laufzeit-Performance alter und neuer Konzepte wird über Benchmarks verglichen. Es wird geklärt, wann parallele Streams verwendet werden sollen, ob die IBM J9 schneller als die HotSpot JVM ist und ob Java oder Scala parallel schneller ist. Über Haskell sollen fehlende funktionale Konzepte von Java aufgedeckt und Alternativen bereitgestellt werden.

Lambda-Ausdrücke und Streams benötigen weniger Code und sind lesbarer und wartbarer. Dafür ist Exception-Handling und Debugging umständlicher. Lambdas werden in der JVM zur Laufzeit über invokedynamic umgesetzt. So werden keine class Dateien erzeugt und die Übersetzungsstrategie kann zukünftig trotz Rückwärtskompatibilität optimiert werden. Die durchgeführten Benchmarks zeigen, dass sequenzielle Streams etwas langsamer als Schleifen sind. Parallele Streams bieten meist einen Performance-Vorteil, sollten jedoch nur bei passenden Quellen, vielen Elementen und komplexen Operationen verwendet werden. Java ist durch Streams für primitive Datentypen wegen Boxing/Unboxing schneller als Scala Collections. Bei funktionalen Konzepten hat Java Nachholbedarf. Es gibt zwar Monaden und eine verzögerte Auswertung bei Streams. Jedoch treten bei Rekursion wegen fehlender Optimierungen Stackoverflows auf. Algebraische Datentypen und Pattern-Matching werden unzureichend unterstützt, können jedoch über Lambdas selbst umgesetzt werden.

Abstract

Parallel programming and cheap multi core processors make functional concepts more popular. That is the reason why object oriented languages take over functional concepts. Hence Java 8 added lambda expressions and streams. Now it is possible to use behaviour as data in form of higher order functions. Streams make it possible to declaratively process data in a sequential or parallel manner. This work aims at showing advantages and disadvantages of these new concepts. With help of examples and decompilation into JVM byte code, we want to analyze the changes and see how other languages implement these concepts. We try to compare the runtime performance of old and new concepts via benchmarks using the tool JMH. There we will answer when to use parallel streams, if the IBM J9 is faster than the HotSpot JVM and if Java or Scala is faster in terms of parallel computations. In looking at the functional concepts of Haskell we want to find missing concepts in Java and present some alternatives.

Lambda expressions and streams need less code and are more readable and maintainable. On the other side, exception handling and debugging is more complicated. Lambdas are translated at runtime with JVMs method call invokedynamic. No class files are generated and Java can optimize their translation strategy in future versions, although Java has backwards compatibility. Our benchmarks show that sequential streams are slightly slower than loops. Parallel streams can achieve performance improvements, but you should only use them with an appropriate stream source, enough elements and complex computations. Java is faster than Scala because it prevents boxing and unboxing with help of specialized streams for primitive data types. Java still needs to catch up on functional concepts. It has monads and supports lazy evaluation on streams. On the other side, support of algebraic data types and pattern matching is insufficient, but it is possible to simulate them via lambda expressions.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Java 8 wurde am 18. März 2014 veröffentlicht. In diesem Zusammenhang wurden nun endlich die von vielen lang ersehnten Lambda-Ausdrücke zur Programmiersprache hinzugefügt. Andere weit verbreitete objektorientierte Sprachen, wie beispielsweise C-Sharp oder C++, hatten Lambda-Ausdrücke bereits wesentlich früher als Sprachkonstrukt hinzugefügt, um so den funktionalen Programmierstil besser zu unterstützen. Ebenso wurden auch bei jenen Sprachen, welche ebenfalls auf der *Java Virtual Machine*(JVM) laufen, bereits Lambdas hinzugefügt. Darunter beispielsweise Scala und Clojure. Da es sich bei Clojure um eine funktionale Programmiersprache handelt, ist es nicht weiters verwunderlich, dass hier Lambda-Ausdrücke Verwendung finden. Scala hat doch eine gewisse Anlehnung an Java und ist trotz vieler funktionaler Ansätze eine objektorientierte Sprache. Erste Überlegungen, um Mechanismen der funktionalen Programmierung in Java miteinzubeziehen, gab es bereits 1997 im Projekt *Pizza*. Anschließend wurde 1997 in Java 1.1 in Form von anonymen inneren Klassen die Möglichkeit gegeben, Code in Form von Daten innerhalb einer Methode zu übergeben. In den Jahren 2006 bis 2009 wurden verschiedene Ansätze für die Umsetzung des Hinzufügens von Lambda-Ausdrücken zu Java diskutiert. Dies war kurz nach der Veröffentlichung von Java 5, wobei unter anderem generische Typen zur Sprache hinzugefügt wurden. Doch diese Ansätze im Bezug auf Lambda-Ausdrücke wurden dann relativ schnell wieder verworfen. Innerhalb der Java-Community war man sich nicht sicher, ob Java jemals über einen offiziellen Release Lambda-Ausdrücke bekommen würde. Meist wurden externe Bibliotheken verwendet, um den funktionalen Programmierstil in Java besser einsetzen zu können, wie beispielsweise *Google Guava* beziehungsweise *Apache Commons*. Im Dezember 2009 war es dann soweit, *OpenJDK Project Lambda* wurde ins Leben gerufen. Im November bekam das Projekt eine eigene JSR-Nummer 335 zugewiesen und wurde von einem offiziellen Java-Commitee bearbeitet.

Die in Java 8 neu hinzugefügten Features erlauben es ProgrammiererInnen nun, funktionale Programmierkonstrukte effizienter verwenden zu können. Dazu zählt unter anderem die

Übergabe von Code als Daten. Damit kann Verhalten an andere Java-Klassen delegiert werden. Diese Delegation ist ein entscheidender Vorteil für ProgrammiererInnen, vor allem auch beim Schreiben neuer Bibliotheken in Java. Bisher musste man als ProgrammiererIn immer beschreiben, was gemacht werden soll, aber zusätzlich dazu auch wie es umgesetzt werden soll. Dies spiegelt sich in der herkömmlichen Iteration über Kollektionen wider. Mit Hilfe der Lambda-Ausdrücke wird es nun möglich nur das, was gemacht werden soll in Form einer anonymen Funktion an andere Bibliotheken zu übergeben. Man übergibt in Form von Funktionen höherer Ordnung jenes Verhalten, das durchgeführt werden soll und überlässt es der jeweiligen Bibliothek selbst, wie das zu erledigende ausgeführt wird. Der Entwickler selbst braucht sich anschließend nur mehr Gedanken darüber zu machen, was erledigt werden soll. Das alles war auch schon vor Java 8 möglich. Jedoch wurde auf Grund des zusätzlich benötigten Standardcodes, welcher auch als Boilerplate-Code bezeichnet wird, den anonyme innere Klassen mit sich bringen, weitgehend auf diese Art der Programmierung verzichtet. Lediglich in Form von so genannten Callback-Interfaces wurde diese Art von Code-als-Daten verwendet. Dabei wurde also Verhalten an jene Objekte weitergegeben, welche als Methodenargument ein Callback-Interface hatte. Hier wurde die Funktionalität jedoch nicht direkt übergeben, sondern gekapselt in einem Objekt, welches extra für diese Funktion angelegt werden musste und eine Implementierung des Callback-Interface darstellt.

Ein weiteres wichtiges Feature, welches zu Java 8 hinzugefügt wurde, sind so genannte Streams. Die Sammlung aller Klassen und Interfaces, die gemeinsam das Konstrukt des Streams erzeugen, werden als *Streams API* bezeichnet. Eine komfortable Verwendung dieser wurde erst durch die neu hinzugefügten Lambda-Ausdrücke ermöglicht. Mit Hilfe von Streams ist es möglich, beispielsweise Daten von Kollektionen über so genannte Bulk-Operationen zu verarbeiten. Dabei werden den Operationen der Streams Lambda-Ausdrücke übergeben. Es handelt sich bei den Operationen auf Streams also um Funktionen höherer Ordnung, da diese als Argumente Funktionen entgegen nehmen. In wie fern diese Operationen dann abgearbeitet werden, wird von den neuen Bibliotheken selbst bestimmt. Man übergibt hier wieder nur, was gemacht werden soll. Im Zusammenhang mit Streams wurde auch die parallele Verarbeitung von Daten in den Vordergrund gerückt. Da die Bibliotheken für die Verarbeitung der Daten zuständig sind, braucht man sich nicht darum zu kümmern, wie genau die Daten parallel abgearbeitet werden. Die Frage, die sich nun in dieser Arbeit stellt, ist, inwiefern Lambda-Ausdrücke die Programmierung in Java vereinfachen werden, aber auch wie und in welchen Situationen die neuen Konzepte anzuwenden sind. Auch in Hinsicht auf die Performance ist die neue Streams API sehr interessant, sequentiell als auch parallel.

1.1 Zielsetzung

Forschungsfrage Vor- und Nachteile des Einsatzes von Lambda-Ausdrücken und Streams in Java 8 gegenüber herkömmlichen Programmiertechniken, mit entsprechenden Konzepten in anderen Sprachen, hinsichtlich Ausdrucksstärke und Effizienz analysieren.

In diese Arbeit sollen die Vor- und Nachteile, die bei der Einführung von Lambda-Ausdrücken in Java 8 zur Sprache hinzugekommen sind, erkannt und analysiert werden. Dabei soll die allgemeine Funktionsweise und der Einsatz von Lambda-Ausdrücken theoretisch und praktisch, anhand von Beispielen näher gebracht werden. Des Weiteren soll die neue Streams API genauer betrachtet werden, dessen Hinzufügen und Verwendung eigentlich erst durch Hinzufügen von Lambda-Ausdrücken sinnvoll geworden ist. Streams sollen beachtliche Vorteile im Bezug auf die Performance, durch die parallele Programmierung und Abarbeitung von Daten ermöglichen. Dabei wollen wir uns den inneren Aufbau von sequenziellen und parallelen Streams in Java genauer ansehen. Ein Vergleich zwischen Streams und Schleifen soll die Vorteile und Nachteile der Verwendung der Streams API näher bringen. Die Performance-Verbesserungen durch die parallele Verarbeitung der Daten sollen anhand von Beispielen und in diesem Zusammenhang durchgeführten Benchmarks verdeutlicht werden. Auch Schwachstellen bei der Abarbeitung von Streams, sei es sequentieller oder paralleler Art, sollen dadurch aufgedeckt werden.

Die neuen Konzepte sollen mit anderen Programmiersprachen verglichen werden. Am Anfang der Arbeit werden die zu vergleichenden Sprachen kurz vorgestellt. Die folgenden Sprachen werden dafür in Betracht gezogen: Java, Scala, Clojure, C-Sharp, C++ und Haskell. Scala und Clojure werden deshalb miteinbezogen, da es sich hierbei um Sprachen handelt, welche ebenfalls wie Java auf der JVM laufen. Gerade der Vergleich zu Scala dürfte hier interessant sein, da hier bereits einige funktionale Konzepte umgesetzt wurden. Daher wird das Hauptaugenmerk im Vergleich zu anderen JVM Sprachen im Vergleich zwischen Java und Scala liegen. Um zusätzlich einen Vergleich mit anderen, häufig verwendeten objektorientierten Sprachen außerhalb der JVM zu gewährleisten, sollen Lambda-Ausdrücke in C-Sharp und C++ betrachtet werden. Anschließend wird Java in Hinsicht auf die durch das Hinzufügen von Lambda-Ausdrücken neu hinzugekommenen Konzepte der funktionalen Programmierung mit der funktionalen Programmiersprache Haskell verglichen. Sollte es funktionale Konzepte geben, die in Java trotz Lambda-Ausdrücken noch nicht direkt unterstützt werden, so sollen alternative Möglichkeiten präsentiert werden. So wollen wir etwa anschauen, welche Möglichkeit Java in Hinsicht auf unveränderbare Datenstrukturen hat und ob beispielsweise das klassische Konzept einer Monade, wie es aus Haskell bekannt ist, auch in Java 8 Verwendung findet.

Im Zusammenhang mit den Lambda-Ausdrücken in Java sollen zunächst die Syntax und sonstige allgemeine Aspekte untersucht werden. Dazu zählt unter anderem auch der Typinferenz-Algorithmus, welcher in Java 8 vermehrt im Zusammenhang mit Lambda-Ausdrücken zum Einsatz kommt. Dadurch wird es ermöglicht, bei Argumenten den Typ wegzulassen, welcher anschließend vom Compiler anhand von zusätzlichen Informationen abgeleitet werden kann. Ebenso soll ein Vergleich zwischen Lambda-Ausdrücken und anonymen inneren Klassen durchgeführt werden. Auch andere neue Features von Java 8, wie beispielsweise Methoden-Referenzen, die einen sehr engen Bezug zu Lambda-Ausdrücken haben, aber auch Default-Methoden und statische Methoden in Interfaces werden betrachtet.

Anschließend soll die Umsetzung der Lambda-Ausdrücke innerhalb der JVM analysiert

werden. Dazu zählt zum Beispiel der bereits in Java 7 hinzugefügte JVM-Call *invokedynamic*. Dieser Befehl wurde damals eigentlich für den Support von dynamisch typisierten Sprachen in der JVM hinzugefügt. Demnach soll die JVM Bytecode-Repräsentation von Lambda-Ausdrücken in Java mit jener von Scala verglichen werden, ebenso sollen Vor- und Nachteile in Hinsicht auf die Performance dargestellt werden. Um die Umsetzung innerhalb der JVM besser nachvollziehen zu können, soll auch die Funktionsweise der Java Virtual Machine und des Befehls *invokedynamic* und der ebenfalls neu hinzugefügten *Method-Handles* näher betrachtet werden.

Mit Hilfe der durchgeführten Benchmarks sollen die Performance-Vorteile durch Streams ausgearbeitet und verdeutlicht werden. Hier wird sich auch zeigen unter welchen Umständen parallele Streams zum Einsatz kommen können und wann man eher darauf verzichten sollte und ob man ab Java 8 nun immer Streams verwenden sollte. Des Weiteren wird sich in den Benchmarks zeigen, welche JVM für die neue Streams-API derzeit besser geeignet ist. Dazu werden alle durchgeführten Benchmarks auf der JVM von Oracle namens *HotSpot*, welche auch als Referenzimplementierung angesehen wird, und auf der J9 von IBM durchgeführt. Außerdem werden wir hier die Laufzeit-Performance zwischen Java und Scala in Hinsicht auf die parallele Verarbeitung von Daten betrachten.

1.2 Methodisches Vorgehen

Zuerst wird eine Literatur-Recherche durchgeführt. Eine sehr interessante Wissensquelle bieten hier die offiziellen Unterlagen und Spezifikationen im Zusammenhang mit *OpenJDK JSR-335*. Der Informationsgehalt der gefundenen Quellen wird anschließend ausgewertet.

Im zweiten Schritt sollen alle relevanten Themen, die sich auf Lambda Ausdrücke beziehen, identifiziert werden. Dazu gehören zum Beispiel funktionale Programmierung und dessen Konzepte im Allgemeinen, Methoden-Referenzen, die in der JVM verwendeten Mechanismen um Lambda-Ausdrücke darzustellen (invokedynamic), allgemeine Funktionsweise der JVM und so weiter. Danach kann zu diesen Themen, in einem größeren Umfang eine Literatur-Recherche betrieben werden.

Um die Funktionsweise von Lambda Ausdrücken besser verstehen und nachvollziehen zu können und auch für den Beitrag an der Diplomarbeit selbst sollen Konzepte der Lambda-Ausdrücke und der Streams API durch kurze Beispiele vorgestellt werden.

Mit Hilfe von javap kann aus dem jeweiligen Java-Code JVM-Bytecode erzeugt werden. Dadurch kann ein Vergleich der Repräsentation der Lambda-Ausdrücke zwischen mehreren JVM Sprachen erläutert werden, wie zum Beispiel ein Vergleich zwischen Java und Scala. Auch kann dadurch der Unterschied zwischen anonymen inneren Klassen und Lambda-Ausdrücken verdeutlicht werden.

Abschließend werden mit Benchmarks die Performance-Unterschiede die mit der neuen Streams API im Vergleich zu anderen Möglichkeiten, in Java parallelen Code auszuführen, verglichen. Dabei wird das Micro-Benchmarking Tool *Java Microbenchmark Harness (JMH)* verwendet. Dadurch können typische Fehler, die bei herkömmlichen Benchmarks

auftreten vermieden werden. Ein typischer Fehler wäre die Messung der Systemzeit vor und nach der Ausführung des zu testenden Codes, da diese Messung ungenau und nicht aussagekräftig wäre.

1.3 Stand der Technik

Java war vor dem Release von Version 8 relativ stark auf objektorientierte Programmierung ausgerichtet. Nun wird jedoch wegen der günstigen Mehrkernprozessoren die parallele Ausführung von Programmen immer relevanter, um beispielsweise auf große Datenmengen Operationen effizient anwenden zu können. Besonders gut eignet sich dafür das Paradigma der funktionalen Programmierung. Grund dafür ist der Aufbau in Form von Funktionen, welche ohne jegliche Seiteneffekte auskommen. Dadurch lässt sich mathematisch leichter über die Programme und deren Ablauf urteilen. Zusätzlich helfen unveränderliche Datenstrukturen dabei funktionale Programme leichter parallel ablaufen zu lassen, da gleichzeitige Lesezugriffe auf Datenstrukturen kein Problem darstellen. Außerdem sind die deklarativen Problemlösungen meist lesbarer und dadurch auch wartbarer als objektorientierte Programme. Die objektorientierte Programmierung verwendet stattdessen Synchronisationsmechanismen um Programme parallel ablaufen zu lassen. Grund dafür sind die standardmäßig verwendeten, veränderbaren Datenstrukturen und gleichzeitiger schreibender Zugriff auf diese. Der imperative Programmierstil ist außerdem schlechter geeignet um Problemlösungen lesbar als Beschreibung des Problems im Code darzustellen. Viele andere objektorientierte Sprachen wie C-Sharp, C++ oder Scala unterstützen bereits länger Lambda-Ausdrücke und funktionale Programmierkonzepte.

Grundlage dieser Arbeit ist das Release von Java 8. Dadurch wurden unter anderem Lambda-Ausdrücke, Methoden-Referenzen und die Streams API zur Sprache hinzugefügt. Auf der OpenJDK-Seite befindet sich unter JSR 335 das Project Lambda, dessen Ziel die Erweiterung von Java um Lambda-Ausdrücke war. Dort befinden sich zwei wichtige Referenzen. In [Goe13a] wird ein Überblick über die sprachlichen Änderungen von Java in Version 8 gegeben. Im zweiten Dokument *State of the Lambda:Libraries Edition*[Goe13b] werden jene Änderungen, die auf Grund der hinzugefügten Lambda-Ausdrücke innerhalb der Bibliotheken stattgefunden haben, aufgelistet. So werden unter anderem interne und externe Iteration beschrieben, ein Überblick über die neue Streams API wird gegeben und in diesem Zusammenhang auch die Möglichkeit, Parallelität besser ausnutzen zu können. Auf der offiziellen Oracle Seite findet man die Änderungen in Java 8 nochmals aufgelistet, des Weiteren gibt es in den Javadocs Informationen zu Lambda-Ausdrücken, Methoden-Referenzen, Default-Methoden und der Streams API.[Orae]

Lambda-Ausdrücke und Methoden-Referenzen können nun anstelle von anonymen inneren Klassen verwendet werden, um Funktionen höherer Ordnung mit weniger Code und dadurch lesbarer und wartbarer umzusetzen. Die Streams API macht es unter Verwendung dieser neuen Konzepte möglich, Daten aus verschiedenen Quellen über mehrere, aneinander gekettete Operationen entweder sequenziell oder parallel zu verarbeiten. Die parallele Verarbeitung wird hierbei von der Streams API selbst gehandhabt. Dabei kann

man beispielsweise Collections aus der Java Collections API als Quelle verwenden. Eine vernünftige Verwendung der Streams API wäre ohne Lambda-Ausdrücke auf Grund der schlechten Lesbarkeit von anonymen inneren Klasse nicht möglich gewesen. Die Problemlösungen mittels Streams API lesen sich auf Grund des Einsatzes von funktionaler Programmierung wie die Beschreibung der Probleme. Als ProgrammiererIn definiert man hier über die Operationen der Streams nur mehr was gemacht werden soll. Die eigentliche Umsetzung übernimmt die API. Dies wird über die Verwendung von Funktionen höherer Ordnung möglich. Bei der parallelen Verarbeitung der Daten durch die Streams API wird das Problem in kleinere Teilprobleme, so genannte Tasks, aufgeteilt. Diese können dann über mehrere Threads, welche sich aus Effizienzgründen innerhalb eines Thread-Pool befinden, auf die Kerne der CPU aufgeteilt und verarbeitet werden. Die Teilergebnisse der Tasks werden anschließend zu einem Endergebnis zusammengefügt. Ermöglicht wird dies durch das Fork-Join-Framework, welches im Hintergrund der parallelen Streams verwendet wird.

Anonyme innere Klassen werden in Java durch zusätzlich erzeugte *class* Dateien realisiert. Dadurch werden bei Verwendung anonymer innerer Klassen jedes Mal zusätzliche Dateien erzeugt. Die JVM unterstützt nach wie vor keine Funktionstypen. Daher werden Lambda-Ausdrücke ebenso wie anonyme innere Klassen durch Klassen dargestellt. Bei den Lambda-Ausdrücken geschieht dies jedoch dynamisch, über die Laufzeit von Java. Hier wird der in Java 7 hinzugekommene JVM Befehl *invokedynamic* verwendet. Dieser wurde für die Unterstützung von dynamisch typisierten Programmiersprachen innerhalb der JVM hinzugefügt. Durch diesen JVM-Befehl wird beim ersten Aufruf dynamisch über eine Bootstrap-Methode eine synthetische innere Klasse erzeugt, welche dann den Lambda-Ausdruck repräsentiert. Dies erzeugt im Gegensatz zu anonymen inneren Klassen keine zusätzlichen *class* Dateien im Dateisystem. Die synthetische Klasse implementiert dabei ein funktionales Interface. Dies ist die neue Bezeichnung für Interfaces, welche genau eine abstrakte Methode definieren. *Invokedynamic* und die Verwendung der Umwandlung der Lambda-Ausdrücke zur Laufzeit eignen sich gut um zukünftig andere Übersetzungsstrategien verwenden zu können. Da Java Versionen immer Rückwärtskompatibel sein müssen, hätte man beispielsweise bei der Verwendung von Method-Handles als Lambda-Ausdrücke in zukünftigen Versionen die Übersetzungsstrategie nicht mehr ändern können. In *Lambda: A peek under the hood* [Goe12a] wird genauer auf die Realisierung von Lambda-Ausdrücken auf JVM-Bytecode-Level eingegangen. Ein weiteres wichtiges Dokument im Bezug auf Java 8 und dessen Umsetzung von Lambda-Ausdrücken in der JVM ist die neue Java Virtual Machine Spezifikation für Java 8. Hier wird sehr ausführlich die Funktionsweise der JVM und invokedynamic beschrieben.[LYBB15]

1.4 Struktur der Arbeit

Kapitel 2 beschreibt das Umfeld. Es soll zuerst ein Vergleich der Programmierparadigmen der objektorientierten und der funktionalen Programmierung durchgeführt werden. Dabei werden die wesentlichen Konzepte beider Paradigmen vorgestellt. Durch das Hinzufügen von Lambda-Ausdrücken zu Java 8 schlägt die objektorientierte Sprache Java ihre Wurzeln

auch in Richtung der funktionalen Programmierung. Abschließend sollen in diesem Kapitel die in dieser Arbeit vorkommenden Programmiersprachen kurz vorgestellt werden. Dabei werden die wichtigsten Konzepte der jeweiligen Programmiersprache präsentiert.

Das Kapitel 3 dient als Einführung in die Lambda-Ausdrücke. Hier werden anonyme innere Klassen vorgestellt und daraus abgeleitet, warum ohne Lambda-Ausdrücke keine effiziente Anwendung von Funktionen höherer Ordnung in Java möglich wären. Es wird die Syntax von Lambda-Ausdrücken präsentiert, gefolgt von den funktionalen Interfaces, die eine sehr wichtige Rolle für Lambdas spielen. Danach wird erklärt, welche Arten von Lambda-Ausdrücken es gibt, die ebenfalls neu hinzugekommenen Methoden-Referenzen und Default- und statische Methoden in Interfaces werden vorgestellt. Methoden-Referenzen werden deshalb erklärt, weil sie sich sehr ähnlich wie Lambda-Ausdrücke verhalten. Default-Methoden und statische Methoden in Interfaces haben die Einführung von Lambda-Ausdrücken in bereits bestehenden Bibliotheken (zum Beispiel Collections API) wesentlich vereinfacht. In diesem Kapitel soll ebenfalls erwähnt werden, welche Software-Entwurfsmuster sich durch die Einführung von Lambda-Ausdrücken geändert haben.

Kapitel 4 beschäftigt sich mit der neuen Streams API. Hier wird zuerst erklärt, was ein Stream ist und wie er sich von der alten Collections API unterscheidet. Nach einer kurzen Einführung wird der interne Aufbau von Streams mit Hilfe von Spliteratoren vorgestellt. Die interne Iteration und Eigenschaften wird mit der externen Iteration verglichen. Es wird darauf eingegangen, in wie fern man in Java Streams erzeugen kann, mit welchen Befehlen man Daten verarbeiten kann und wie diese Daten anschließend in verschiedene Formen gebracht werden können.

In Kapitel 5 wird der Aufbau von Lambda-Ausdrücken innerhalb der JVM vorgestellt. Zuerst wird die Funktionsweise und eine kurze geschichtliche Einführung zur JVM näher gebracht. Dazu gehören unter anderem die Datenbereiche und der Befehlssatz der JVM. Des Weiteren wird erklärt wie Java-Code auf der JVM ausgeführt werden kann. Danach wird der bereits in Java 7 hinzugefügte Befehl *invokedynamic*, der eine sehr wichtige Rolle bei der Umsetzung der Lambda-Ausdrücke in Java 8 spielt, genauer vorgestellt. In diesem Zusammenhang werden auch Method-Handles erklärt. Anschließend wird der Bytecode von anonymen inneren Klassen mit jenem Bytecode der Lambda-Ausdrücke verglichen und analysiert. Ebenso wird die Umsetzung von Methoden-Referenzen innerhalb der JVM betrachtet. Der interne Aufbau von Lambda-Ausdrücken in Java wird mit jenen der JVM-Sprachen Clojure und Scala verglichen, wobei es hier Unterschiede zwischen den Scala Versionen 2.11 und 2.12 gibt.

Kapitel 6 zeichnet sich durch den Vergleich von Java in Hinsicht auf Lambda-Ausdrücke und funktionale Konzepte mit anderen Programmiersprachen aus. Dazu wird betrachtet, wie Lambda-Ausdrücke in C-Sharp und C++ umgesetzt wurden. Auf Seite der JVM-Sprachen wird ein allgemeiner Vergleich zwischen Java und Scala in Hinsicht auf einige funktionale Konzepte vorgestellt. Ebenso werden Unterschiede von Collections und Streams zwischen Java und Scala besprochen.

Kapitel 7 beschäftigt sich mit der Laufzeit-Performance der Lambda-Ausdrücke, und

in diesem Zusammenhang liegt das Hauptaugenmerk auf der Performance der neu hinzugefügten Streams API. Zuerst soll in der Bytecode-Repräsentation verglichen werden, ob anonyme innere Klassen oder Lambda-Ausdrücke schneller ablaufen. Dazu werden bereits von Oracle durchgeführte Benchmarks zur Hand genommen. Bei der Stream-Performance werden anschließend die folgenden Vergleiche mittels Microbenchmarks über JMH durchgeführt. So soll einerseits über die Benchmarks erkannt werden, welche Streams von verschiedenen Quellen am besten für parallele Verarbeitung geeignet sind, und wann man sequentielle oder parallele Streams verwenden sollte. Es sollen auch Fälle aufgezeigt werden, sofern es diese gibt, bei denen der Einsatz eines Streams die Performance nicht verbessert. Ein weiterer interessanter Vergleich ist die Geschwindigkeit bei der Verarbeitung von Daten über Streams und Collections in Java und Scala. Zusätzlich dazu soll die Performance zwischen zwei unterschiedlichen virtuellen Maschinen getestet werden. Dabei wird einerseits die JVM von Oracle beziehungsweise von OpenJDK verwendet, welche als Referenzimplementierung bezeichnet wird. Andererseits wird die JVM von IBM verwendet.

In Kapitel 8 werden einige Konzepte der funktionalen Programmiersprache Haskell mit Java 8 verglichen. Dabei soll überprüft werden, welche Konzepte von Haskell sich nun durch Hinzufügen von Lambda-Ausdrücken einfacher in Java 8 umsetzen lassen, beziehungsweise welche bereits umgesetzt wurden und welche nach wie vor einiges an zusätzlichem Programmieraufwand benötigen. So sollen hier beispielsweise Pattern-Matching und Monaden untersucht werden. Sofern Java 8 keine direkte Möglichkeit für die Umsetzung der jeweiligen funktionalen Konzepte bietet, sollen hier Alternativen vorgestellt werden.

KAPITEL 2

Paradigmen und Programmiersprachen

Dieses Kapitel soll als Einführung dieser Arbeit dienen. Dazu soll das objektorientierte Paradigma mit jenem der funktionalen Programmierung verglichen werden. Anschließend werden die in dieser Arbeit erwähnten Programmiersprachen kurz vorgestellt. Einige Konzepte von Scala werden in einem späteren Kapitel dieser Arbeit genauer vorgestellt, ebenso verhält es sich bei Haskell.

2.1 Vergleich zwischen objektorientierter und funktionaler Programmierung

In diesem Unterkapitel sollen die Programmierparadigmen der objektorientierten Programmierung mit jener der funktionalen Programmierung miteinander verglichen werden. Dieser Vergleich soll dazu dienen, die Vorteile des Hinzufügens von Lambda-Ausdrücken zu objektorientierten Sprachen zu verdeutlichen. Dieses neue Sprachkonzept bedeutet jedoch nicht, dass Java 8 ab sofort oder in Zukunft zu einer rein funktionalen Programmiersprache werden soll. Es können jedoch die Vorteile beider Programmierparadigmen für eine effizientere Entwicklung und Wartung der Software eingesetzt werden. Des Weiteren bietet dieses Unterkapitel eine gute Grundlage für die anschließende Erläuterung von Lambda-Ausdrücken in den nächsten Kapiteln.

2.1.1 Objektorientierte Programmierung

Hier soll ein kurzer Überblick über das objektorientierte Programmierparadigma gegeben werden. Wie der Name schon sagt, spielen bei der objektorientierten Programmierung Objekte, gemeinsam mit Klassen eine wichtige Rolle. Die objektorientierte Programmierung zeichnet sich, sofern dessen Konzepte korrekt angewandt wurden, für seine

hohe Wartbarkeit und gute Veränderbarkeit der Software aus. Bei der objektorientierten Programmierung geht es vorallem darum, die echte Welt und dessen Interaktion zwischen verschiedenen Elementen zu modellieren.[Pun07]

Die grundlegende Einheit bildet das Objekt. Objekte sind Instanzen von Klassen, wobei die jeweilige Klasse die Eigenschaften eines Objektes beziehungsweise dessen Grundstruktur festlegt. Während der Ausführung besteht das Programm aus einer Menge von Objekten. Die Klassen beinhalten Variablen und Methoden. Zugriffsberechtigungen beziehungsweise Sichtbarkeit spielen eine wichtige Rolle in der objektorientierten Programmierung. So gibt es Variablen und Methoden auf die von außen zugegriffen werden kann, diese werden in Java durch das Schlüsselwort *public* angegeben. Andere Methoden und Variablen können nur von der Klasse selbst verwendet werden. Ein Objekt kann eine Methode eines anderen Objekts mit Hilfe einer Nachricht aufrufen. Das bedeutet, dass die Objekte mittels Nachrichten miteinander kommunizieren können.

Objekte zeichnen sich durch die folgenden Eigenschaften aus. Sie besitzen eine Identität, einen Zustand, Verhalten und eine Schnittstelle. Die Identität ist die eindeutige Kennzeichnung eines Objektes. Der Zustand eines Objekts wird über die Werte der Variablen festgelegt. Das Verhalten und die Schnittstelle des Objektes legen fest, wie das jeweilige Objekt auf eine Nachricht von außen reagiert. Das Verhalten ist abhängig von der aufgerufenen Methode, dessen Parametern und dem Zustand des Objektes, also den Werten der Variablen des Objekts. Datenabstraktion spielt ebenfalls eine wichtige Rolle in der objektorientierten Programmierung. Sie wird durch die Kapselung der Daten und durch das Verbergen dieser erreicht. Beim objektorientierten Paradigma wird versucht, die Objekte möglichst unabhängig voneinander zu halten. Dies wird durch eine schwache Kopplung der Objekte und eine gute Kapselung der Daten ermöglicht.

Polymorphismus ist ebenso eine weit verbreitetes Konzept. Polymorphismus bedeutet, dass jede Variable und jede Routine gleichzeitig mehrere Typen haben kann. So wird meist zwischen den folgenden Typen unterschieden. Variablen beziehungsweise Routinen haben einen deklarierten Typ. Das ist jener Typ, welcher der jeweiligen Variable/Routine zugewiesen wurde. Der statische Typ wird vom Compiler ermittelt; er kann für compilerbasierte Programmoptimierungen verwendet werden. Der dynamische Typ ist der spezifischste Typ einer Variable oder einer Routine. Er wird während der Laufzeit für Typüberprüfungen herangezogen.

Ein weiteres wichtiges Konzept in der objektorierten Programmierung ist Vererbung. Neue Klassen werden von bestehenden Klassen abgeleitet. Dabei können bei den meisten objektorientierten Programmiersprachen die Unterklassen erweitert und Methoden der Oberklasse überschrieben werden. Auch Interfaces sind ein beliebtes Konzept objektorientierter Programmiersprachen. Dabei werden Schnittstellen für die jeweiligen Klassen, welche das Interface implementieren, festgelegt.[AC12]

2.1.2 Funktionale Programmierung

Das Interesse für funktionale Programmierung stieg mit der Zunahme an Multikern-Prozessoren. Auf Grund der positiven Eigenschaften der funktionaler Programmierung, welche in diesem Unterkapitel noch besprochen werden, wird die parallele Programmierung für Entwickler wesentlich vereinfacht. Dadurch sollen robuste, nebenläufige Programme mit hoher Wiederverwendbarkeit ermöglicht werden. Ebenso fällt bei der Bearbeitung von großen Datenmengen im Vergleich zur objektorientierten Programmierung der Overhead für das Erzeugen der Objekte weg. Somit ist funktionale Programmierung auch bei der Verarbeitung von größeren Datenmengen zunehmend interessanter geworden. Im Folgenden wird nun geklärt, was funktionale Programmierung ausmacht und welche wichtigen Prinzipien dieses Programmierparadigma beinhaltet.[Wam11]

Das funktionale Paradigma basiert auf dem Lambda-Kalkül. Kurz gesagt handelt es sich beim Lambda-Kalkül um eine formale Sprache für Funktionen und deren Auswertung. Bei der funktionalen Programmierung besteht jedes Programm aus einer oder mehreren Funktionen. So kann beispielsweise eine Main-Funktion für die Programmausführung verantwortlich sein. Diese nimmt Argumente in Form von Parametern entgegen und liefert nach Berechnung ein Resultat zurück. Die Funktion kann ihrerseits eine oder mehrere andere Funktionen aufrufen. Funktionen, die keine anderen, vom Programmierer definierten Funktionen mehr aufrufen, werden aus Sprachprimitiven der jeweiligen funktionalen Sprache aufgebaut.[Hug89]

Die Funktionen innerhalb von funktionalen Programmiersprachen verhalten sich meist wie mathematische Funktionen, sofern es sich um pure funktionale Programmiersprachen handelt. Da es keine destruktive Zuweisung von Variablen gibt, können sich Werte der Variablen während der Ausführung des Programms nicht ändern. Ebenso wird bei funktionalen Sprachen auf Seiteneffekte verzichtet, es wird lediglich das Resultat der jeweiligen Funktion berechnet. Auf Grund dieser Eigenschaften können bei funktionalen Programmen mathematische Beweise für die Korrektheit des Programmes einfacher gefunden werden. Ein weiterer Vorteil ist es, dass die Reihenfolge, in der die Funktionen ausgeführt und ausgewertet werden, nicht relevant ist. Ebenso erhält man bei einer mathematischen Funktion mit denselben Eingabeparametern immer dasselbe Ergebnis. Dadurch können die Funktionen durch ihr Resultat ersetzt werden. Diese Eigenschaft wird als referenzielle Transparenz bezeichnet. Über die referenzielle Transparenz können Programme beziehungsweise Funktionen mathematisch einfach nachvollzogen werden. Beim funktionalen Programmierparadigma wird die Reihenfolge der Auswertung nicht durch den Programmierer bestimmt. In funktionalen Programmiersprachen wird statt Schleifen Rekursion eingesetzt.[Hug89]

Bei *guter* Software spielt Modularisierung eine entscheidende Rolle. Kleinere Module können vom Programmierer unabhängiger, einfacher und teilweise schneller programmiert werden. Allgemeine Module können für wiederkehrende Aufgaben ohne Probleme wiederverwendet werden, und auch das Testen wird durch die Module erleichtert. Bei der Modularisierung der Programme geht es darum, ein zu lösendes größeres Problem in

kleinere Teilprobleme aufzuteilen. Diese Teilaufgaben gilt es dann zu lösen und mit Hilfe von deren Ergebnissen kann anschließend das Hauptproblem gelöst werden. Dazu wird jedoch ein Mechanismus benötigt, um die gelösten Teilaufgaben effizient zusammenfügen zu können.[Hug89]

Die funktionale Programmierung bietet dazu zwei mächtige Werkzeuge an. Zum einen wird eine Möglichkeit bereitgestellt, um Funktionen miteinander zu verbinden. Dieses Konzept wird auch bei der neuen Streams API von Java 8 verwendet, um damit große Datenmengen einfach verarbeiten zu können. Dieses Konzept wird als Funktion höherer Ordnung bezeichnet. Im Vergleich zu Funktionen erster Ordnung, die auch bei herkömmlichen objektorientierten Sprachen zum Einsatz kommen, können bei Funktionen höherer Ordnung Funktionen selbst als Parameter angegeben beziehungsweise als Resultat zurückgeliefert werden. Mit Funktionen höherer Ordnung lassen sich Probleme allgemeiner beschreiben und anschließend für die Probleme wiederverwenden. Für ein spezifisches Problem werden dann einfach die jeweiligen Funktionen, die für die Problemlösung benötigt werden, an die wiederverwendbare Funktion höherer Ordnung übergeben. Des Weiteren kann Verhalten in Form von Code-als-Daten an die Funktionen übergeben werden.[Hug89]

Das zweite Werkzeug, welches die Modularisierung in funktionalen Programmiersprachen wesentlich erleichtert, ist die so genannte verzögerte Auswertung. Diese ist auch als *lazy evaluation* bekannt. Diese wird für das Zusammenbauen von Programmen verwendet. Auch davon macht die neue Streams API in Java 8 Gebrauch. Wenn wir beispielsweise zwei Funktionen f und g miteinander kombinieren wollen, also wenn der Input von Funktion f das Resultat von Funktion g darstellt, funktioniert das bei den funktionalen Programmiersprachen, die eine verzögerte Auswertung verwenden, wie folgt. Die beiden Funktionen laufen synchronisiert zueinander ab. Funktion g liefert nur so lange Resultate, so lange Funktion f Daten für die Eingabe benötigt. Durch diese Art der Auswertung können auch unendlich große Datenströme verarbeitet werden.[Hug89]

2.2 Überblick über die Programmiersprachen

Obwohl das Hauptaugenmerk im Bezug auf diese Diplomarbeit bei der Analyse der Lambda-Ausdrücke innerhalb der JVM und speziell auf Java 8 ausgelegt ist, werden auch andere Programmiersprachen, insbesondere auf deren Umsetzung beziehungsweise Implementierung von Lambda-Ausdrücken im Vergleich zu Java betrachtet. In diesem Kapitel soll eine kurze Einführung der zu vergleichenden Sprachen gegeben werden. Dazu zählt einerseits die geschichtliche Evolution der jeweiligen Sprache, aber auch eine Auflistung der wichtigsten Konzepte dieser. Auf Seite der JVM werden die Sprachen Java, Scala und Clojure näher betrachtet. Um aber auch einen Vergleich mit Sprachen außerhalb der JVM zu bekommen, werden zusätzlich noch die beiden weit verbreiteten Sprachen C-Sharp und C++ im Bezug auf Lambda-Ausdrücke angeschaut. Um zusätzlich dazu noch eine Referenz und einen Vergleich mit einer puren funktionalen Sprache zu erhalten, wird die funktionale Programmierung in Haskell ebenfalls besprochen.

Wir beginnen zunächst mit einer Einführung der drei JVM-Sprachen, welche für diese Diplomarbeit im Bezug auf Lambda-Ausdrücke relevant sind. Dazu gehören Java, Scala und Clojure, wobei auf den Vergleich zwischen Java und Scala in einem späteren Kapitel dieser Arbeit genauer eingegangen wird. Bei Clojure wird im Kapitel *Java Virtual Machine* lediglich der interne Aufbau der Lambda-Ausdrücke betrachtet. Genaueres zur Funktionweise der JVM folgt in einem späteren Kapitel dieser Arbeit.

2.2.1 Java

Das *White Paper* zu Java mit dem Titel *The Java language environment*[GM95] wurde im Jahr 1995 unter anderem von James Gosling veröffentlicht. Er gilt als Erfinder von Java und hat damals die Programmiersprache gemeinsam mit *Sun Microsystems* veröffentlicht. Das Unternehmen wurde letztlich im Jahr 2010 von *Oracle* übernommen. Damals wurde Java noch als rein objektorientierte Programmiersprache der Öffentlichkeit vorgestellt:

„Java has no functions. Object-oriented programming supersedes functional and procedural styles. Mixing the two styles just leads to confusion and dilutes the purity of an object-oriented language. Anything you can do with a function you can do just as well by defining a class and creating methods for that class.“[GM95, Seite 28]

JDK 1.0 wurde am 23 Januar 1996 veröffentlicht. In JDK 1.1 (Februar 1997) wurden innere Klassen hinzugefügt. Die Collections API wurde mit Version 1.2 im Dezember 1998 eingeführt. Mit Version 1.3 (Mai 2000) wurde die HotSpot JVM verwendet. Die im Februar 2002 veröffentlichte Version 1.4 erhält für diese Arbeit keine relevanten Erweiterungen. Mit 1.5 wurde die Versionierung von Java auf 5 geändert. Hier erhielt Java im September 2004 mehrere für diese Arbeit relevante Verbesserungen.[Oraj]

- Generische Typen: Typ-Variablen, bieten unter anderem Typ-Sicherheit für Collections.
- Enums Summen-Typen (algebraische Datentypen) für Java.
- for-Each Schleife: for-Schleife wobei über alle Elemente einer Kollektion iteriert wird. Dadurch erspart man sich die direkte Verwendung von Iteratoren.
- java.util.concurrent: Paket um parallele Programmierung in Java zu unterstützen.

Java 6 wurde im Dezember 2006 herausgebracht und enthielt unter anderem Bug-Fixes und Performance-Verbesserungen. Mit Java 7, welches im Juli 2011 erschien, wurde das für Lambda-Ausdrücke wichtige *invokedynamic* zur JVM hinzugefügt. Des Weiteren wurde das Fork-Join-Framework zur Standardbibliothek hinzugefügt. Dieses ist besonders für parallele Streams und für parallele Berechnungen relevant. Außerdem wurde beispielsweise der Diamant-Operator hinzugefügt. Dadurch wurde eine vereinfachte Typ-Inferenz im Zusammenhang mit Generizität möglich.[Oraj]

```
List<Integer> liste = new ArrayList<Integer>();
List<Integer> liste = new ArrayList<>();
```

Java 8 wurde am 18. März 2014 veröffentlicht.

2.2.2 Scala

Scala wurde 2001 in der Schweiz an der *École polytechnique fédérale de Lausanne*, kurz EPFL, unter der Leitung von Martin Odersky entwickelt. Dieser war bereits am Java-Projekt Pizza aus dem Jahr 1997 beteiligt. Bei dieser Erweiterung von Java sollten funktionale Aspekte, beispielsweise Lambda-Ausdrücke und algebraische Datentypen zur Sprache hinzugefügt werden[OW97]. Im Januar 2004 folgte die Veröffentlichung von Scala auf der JVM. Im Juni desselben Jahres wurde Scala für die .NET Plattform freigegeben. Der offizielle Support für die .NET Plattform wurde jedoch bereits eingestellt. Die zweite Version von Scala wurde im März 2006 veröffentlicht. Zum Zeitpunkt des Verfassens dieser Arbeit befindet sich Scala in Version 2.11.[OAC+04a]

Scala vereint Konzepte der objektorientierten Programmierung mit jenen der funktionalen Programmierung. Es handelt sich dabei um eine statisch typisierte Sprache mit einem starken Typsystem. Das bedeutet, dass die Typen der Objekte schon vor der Ausführung dem Compiler bekannt sind. Die Grundidee bei Scala war es, eine Programmiersprache zur Verfügung zu stellen, bei der die Software mit Hilfe bereits vorher definierter Komponenten erstellt wird und nicht wie meistens in der Realität komplett neu aufgebaut werden muss. Komponenten sind in diesem Sinne Teile der Software, die anschließend zu einem größeren Ganzen zusammengebaut werden können. Dabei können verschiedene Komponenten, zum Beispiel Klassen, Bibliotheken oder Frameworks mit Hilfe verschiedener Techniken zusammengefügt werden. Die Sprache wurde deshalb ins Leben gerufen, weil laut Entwickler von Scala, Sprachen wie Java und C-Sharp nicht ausreichend Möglichkeiten für Komponenten-Abstraktion und Komposition von Komponenten bieten.

Scala basiert auf zwei Grundlagen. Die erste Grundlage ist es, eine skalierbare Software zu ermöglichen, wobei unabhängig von der Größe der jeweiligen Software-Teile, jeweils das gleiche Repertoire an Konzepten für die Umsetzung derer verwendet werden soll. Dazu sollen für den Aufbau der Sprache Scala-Mechanismen für Abstraktion, Komposition und Dekomposition von Komponenten angeboten werden. Die zweite Grundlage legt fest, dass die hohe Skalierbarkeit der Kompontenten, welche dann ein Programm bilden, nur über eine Mischung aus objektorientierter und funktionaler Programmierung zu erreichen ist.

Im Vordergrund steht bei Scala eine gute Kompatibilität zu den Sprachen Java und C-Sharp. Scala bedient sich einer sehr ähnlichen Syntax bezüglich der beiden genannten Sprachen. Ebenso wurden viele Aspekte der Typsysteme von Java und C-Sharp übernommen. Dadurch werden Interaktionen mit Java-Programmen innerhalb von Scala ermöglicht. So können beispielsweise Java-Methoden aufgerufen werden. Ebenso können Klassen in Scala von Java-Klassen erben beziehungsweise Interfaces implementieren.[OAC+04a]

Operatoren, Datentypen und Kontrollstrukturen wurden weitgehend von Java und C-Sharp übernommen. In einigen Fällen ist Scala sogar C-Sharp ähnlicher, zum Beispiel bei Generizität. Bezüglich der Objektorientiertheit der Sprache ist Scala eine pure objektorientierte Sprache. Das bedeutet jedoch nicht, dass nur das objektorientierte Paradigma innerhalb der Sprache angewandt werden kann. Es bezieht sich darauf, dass alle Werte, die innerhalb von Scala verwendet werden, als Objekte dargestellt sind. Es gibt also keine primitiven Typen (*int, char, long*) wie in Java. Im Bezug auf die Objektorientiertheit verhält es sich bei Scala wie bei Smalltalk. Alles wird als Objekt dargestellt und jede Operation repräsentiert eine Nachricht, die an ein Objekt gesendet wird. Im Bezug auf das funktionale Programmierparadigma werden durch die Verwendung von Funktionen als *Bürger erster Klasse* die Anwendung von Konzepten der funktionalen Programmierung in Scala vereinfacht. *Bürger erster Klasse* bezieht sich hier darauf, dass Funktionen in Variablen abgespeichert werden können, als Methodenargumente an Methoden übergeben werden können oder aber als Resultat von Methoden zurückgeliefert werden.

Der Vergleich von Java mit Scala im Bezug auf Lambda-Ausdrücke wurde deshalb gewählt, weil Scala wesentlich schneller bei der Umsetzung und beim Einsatz von funktionalen Konzepten innerhalb der JVM agiert hat. Dabei muss jedoch erwähnt werden, dass Java besonders viel Wert auf Abwärtskompatibilität legt und daher einige Entscheidungen langfristig von Bedeutung sind[Orac]. Durch die Mischung aus objektorientiertem und funktionalen Programmierparadigma wurde Scala in den vergangen Jahren zu einer beliebten Sprache innerhalb der JVM.

2.2.3 Clojure

Clojure ist eine dynamisch typisierte, funktionale Programmiersprache, welche für die JVM-Plattform entwickelt wurde. Der Einsatz ist jedoch nicht auf die JVM-Plattform beschränkt, so werden auch die Common Language Runtime (CLR) von Microsoft's .NET Framework und die Javascript-Engine unterstützt. Clojure Version 1.0 wurde am 5. Mai 2009 veröffentlicht. Es vereint die Vorteile einer Skript-Sprache, nämlich eine einfache Zugänglichkeit und die dynamische Entwicklung, mit einer robusten und effizienten Programmierumgebung für parallele Programmierung. Bei Clojure handelt es sich um einen LISP-Dialekt. Daher bezieht sich Clojure auf das Lambda-Kalkül und die Sprache besitzt einen sehr minimalen Kern. Eines der wichtigsten Eigenschaften der LISP-Dialekte ist es, Code als Daten einzusetzen. Es handelt sich bei Clojure um eine kompilierte Sprache, wobei der Programmcode, typisch für Sprachen die in der JVM laufen, in JVM-Bytecode umgewandelt wird.[Cloa][Hic08]

Clojure bietet standardmäßig Speicherstrukturen an, welche nicht verändert werden können. Dadurch wird die parallele Programmierung wesentlich unterstützt und vereinfacht, da der mehrfache Zugriff für Leseoperationen kein Problem darstellt. Für den Fall, dass doch veränderbare Datenstrukturen für die Umsetzung eines Programmes benötigt werden, bietet die Sprache *Software Transactional Memory* und *Reactive Agents* an. Dadurch wird ein sauberes Design beim Aufbau von Programmen für Mehrkern-Prozessoren

sichergestellt, sodass auch bei gleichzeitigem schreibendem Zugriff auf Speicherstrukturen keine Fehler auftreten.[Cloa]

2.2.4 C++

C++ erschien im Jahr 1985 und wurde von Bjarne Stroustrup entwickelt. C++ versteht sich als Weiterentwicklung von C und wird vorwiegend für systemnahe Programmierung verwendet. Unter Systemprogrammierung versteht man die Erstellung von Programmen, die direkt auf Hardware-Ressourcen zugreifen beziehungsweise Ressourcen-limitiert sind. Die zwei wichtigsten Design-Prinzipien von C++ beziehen sich auf das hardwarenahe Level der Programmiersprache. So soll keine andere Sprache, außer Assembler, besser für systemnahe Programmierung geeignet sein. Des Weiteren soll bei Sprachfeatures und fundamentalen Abstraktionen kein Speicherplatz und keine CPU-Zyklen im Vergleich zu Alternativen verschwendet werden. Das bedeutet, dass die Sprachstrukturen von C++ und die Abstraktionen so effizient implementiert sein sollen, dass keine Notwendigkeit durch einen Programmierer entsteht, sich selbst effizientere Alternativen zu programmieren. C++ unterstützt die Prinzipien der generischen, prozeduralen und objektorientierten Programmierung und der Datenabstraktion. Lambda-Ausdrücke wurden mit C++11 zur Sprache hinzugefügt. C++ beinhaltet keine Garbage-Collection, daher muss sich der Programmierer selbst um die Speicherplatzbereinigung kümmern, wenn Variablen und Objekte nicht mehr benötigt werden. Vergisst man auf die Destruktion von nicht mehr benötigten Speicherobjekten, so kann es zu Speicher-Leaks kommen, die dazu führen, dass das Programm nicht mehr ausreichend Speicher für die korrekte Ausführung zur Verfügung hat.[Str13]

2.2.5 C-Sharp

Bei C-Sharp handelt es sich um eine statisch typisierte, objektorientierte Programmiersprache, wobei eine der ersten Implementierungen im Juli 2000 als Teil des *.NET Frameworks* von Microsoft vorgestellt wurde. Die Sprache ist stark an Java angelehnt. So sind unter anderem die Syntax und das Typsystem sehr ähnlich aufgebaut. C-Sharp 1.0 wurde im Jahr 2002 veröffentlicht. Bereits seit dieser Version werden Funktionen als Bürger erster Klasse in Form von Delegaten in der Sprache umgesetzt. Diese sind von der Funktionsweise her ähnlich wie Funktionszeiger in C++, sie sind jedoch in deren Verwendung typsicher. Dabei wird auf eine Methode mit einer vorgegebenen Signatur referenziert. Im Vergleich zu Java gibt es bereits seit der ersten Version eine for-Each Schleife. C-Sharp unterstützt außerdem das Überladen von Operatoren und es werden benutzerdefinierte Werttypen ermöglicht. In C-Sharp 2.0 wurden anonyme Delegaten, Iteratoren und generische Datentypen zur Sprache hinzugefügt. Lambda-Ausdrücke wurden schließlich im April 2010 in der Version 3.0 in die Sprache integriert. Gleichzeitig wurde hier auch das Pendant zur Streams API namens LINQ implementiert.[Ske13] Im Jahr 2017 wurde die aktuelle Version 7.0 veröffentlicht. Hier wurden unter anderem Pattern-Matching und Tupel hinzugefügt. Außerdem ist es direkt möglich, Lambda-Ausdrücke

als Konstruktoren, Finalizer und Getter/Setter zu verwenden, nachdem diese in Version 6.0 bereits zur Deklaration von Methoden verwendet werden konnten.[Tor]

2.2.6 Haskell

Haskell ist eine rein funktionale Programmiersprache. Das bedeutet, dass jedes Programm als mathematische Funktion angesehen wird. Diese Funktion erhält Eingabeparameter und liefert bei der Ausführung mit denselben Parametern immer dasselbe Ergebnis zurück. Dadurch werden Seiteneffekte bei der Ausführung eines Programmes vermieden. Es wird lediglich das Programm mit bestimmten Eingabeparametern aufgerufen und über die Ausführung einer oder mehrerer Funktionen wird ein Resultat berechnet und zurückgeliefert. Die meisten anderen funktionalen Programmiersprachen erlauben dennoch Seiteneffekte und sind daher keine rein funktionalen Sprachen. In Haskell werden solche Seiteneffekte, wie zum Beispiel Eingabe- Ausgabe-Operationen, mit Hilfe von Monaden umgesetzt. Daher handelt es sich bei Haskell, trotz der Möglichkeit von Seiteneffekten, um eine rein funktionale Programmiersprache.[Has]

Die Entwicklung von Haskell wurde 1987 auf einer Konferenz beschlossen, nachdem bereits eine Vielzahl unterschiedlicher funktionaler Programmiersprachen im Umlauf war. Bis 1997 wurden die Versionen 1.0 bis Version 1.4 veröffentlicht. Anschließend wurde im Februar 1999 ein Standard namens Haskell 98 festgelegt. Die aktuelle Version von Haskell ist Haskell 2010. [M+10]

Die Sprache basiert auf dem Lambda Kalkül und es handelt sich um eine polymorphe, statisch typisierte Programmiersprache mit verzögerter Auswertung. Benannt wurde sie nach dem Logiker Haskell Brooks Curry. Die Sprache wurde unter anderem für die Lehre und für die wissenschaftliche Forschung am funktionalen Programmierparadigma ins Leben gerufen. Dennoch sollte der Einsatz der Sprache für die Programmierung großer Systeme nicht vorenthalten bleiben. Zu den Hauptkonzepten von Haskell zählen Funktionen höherer Ordnung, benutzerdefinierte algebraische Datentypen, Pattern-Matching sowie Monaden.[Has]

Anhand von Haskell soll innerhalb dieser Arbeit verdeutlicht werden, welche funktionalen Programmieraspekte in Java 8 nun unterstützt werden und welche Konzepte nach wie vor, ohne weitere Bibliotheken beziehungsweise ohne das Hinzufügen zusätzlicher Konstrukte zu Java, nicht, oder nur mit erhöhtem Aufwand möglich sind.

KAPITEL 3

Lambda-Ausdrücke

In diesem Kapitel sollen die grundlegenden Informationen im Bezug auf Lambda-Ausdrücke vermittelt werden. Zuerst werden einige Hintergrundinformationen bereitgestellt. Da es doch relativ lange im Vergleich zu anderen weit verbreiteten objektorientierten Programmiersprachen gedauert hat, bis Lambda-Ausdrücke ihren Weg in Java 8 gefunden haben, wird zuerst erläutert, wie es zu Projekt Lambda und dessen zusammenhängende Erweiterung von Java um Lambda-Ausdrücke gekommen ist. Anschließend wird erklärt, wie es vor Java 8 möglich war, Code in Form von Daten, also Verhalten, an Methoden zu übergeben. Danach werden die Syntax von Lambda-Ausdrücken und die Eigenschaften von Lambdas im Vergleich zu anonymen inneren Klassen genauer erklärt. In diesem Zusammenhang werden funktionale Interfaces, Methoden-Referenzen, statische und Default-Methoden von Interfaces ebenfalls vorgestellt.

3.1 Lambda-Ausdrücke: Einführung

Die Einführung von Lambda-Ausdrücken zu Java in Version 8 wurde unter dem Projektnamen *Lambda Project* im OpenJDK implementiert. Bei OpenJDK handelt es sich um eine Plattform, über die gemeinsam an einer Open-Source Implementierung von Java gearbeitet wird[Orah]. Die Spezifikation dafür wurde unter der JSR-Nummer 335 festgelegt. Mit Hilfe dieses Projekts wurden endlich Lambda-Ausdrücke zu Java hinzugefügt. Durch das Hinzufügen dieses Sprachkonstruktes ist es nun möglich, Programmcode in Form von Daten über eine einfache Syntax zu übergeben. Lambda-Ausdrücke alleine reichen jedoch noch nicht aus, um das volle Potenzial der Übergabe von Funktionen als Methodenparameter auszuschöpfen. Deshalb wurden des Weiteren die folgenden Sprach-Features zu Java 8 hinzugefügt. Die neuen Sprachkonzepte werden in diesem Kapitel genauer betrachtet.

- Lambda-Ausdrücke,

- Funktionale Interfaces,
- Methoden-Referenzen und Konstruktur-Referenzen,
- Erweiterung der Ziel-Typen und Verbesserung der Typ-Inferenz,
- Default-Methoden und statische Methoden in Interfaces.

3.2 Anonyme innere Klassen

Die Möglichkeit, Funktionen an Methoden zu übergeben wurde nicht erst mit Java 8 hinzugefügt, sondern war bereits davor möglich. So konnten bereits ab Java 1.1, welches am 19. Februar 1997 veröffentlicht wurde, innere Klassen dazu verwendet werden, um Verhalten in Form von Code-als-Daten an Methoden zu übergeben. Da die inneren Klassen keine Namen haben, spricht man auch von anonymen inneren Klassen. Hier hatte man den Vorteil, dass man außerhalb keine neuen Klassen anlegen musste, daher war die Verwendung von anonymen inneren Klassen im Vergleich zu herkömmlichen Klassen komfortabler[Oraf]. Im Gegensatz zu funktionalen Programmiersprachen, wird das Verhalten nicht direkt als Funktion an die jeweilige Methode übergeben. Stattdessen wird ein Objekt einer Klasse an die Methode übergeben. Die Klasse implementiert eine Schnittstelle, welche über ein *Interface* festgelegt wurde. In dieser Schnittstelle wird festgelegt, dass eine bestimmte Funktion implementiert werden muss. So genannte Callback-Interfaces wurden seither für die Kapselung einer einzigen Funktion verwendet. Die funktionale Programmierung gewinnt auf Grund der Mehrkernprozessoren immer mehr an Bedeutung. Daher spielten die Callback-Interfaces bereits für die Verwendung innerhalb paralleler Java-Bibliotheken eine wichtige Rolle und wurden vermehrt eingesetzt. Einige wichtige Callback-Interfaces vor Java 8 werden anschließend aufgelistet.[Orab]

- java.lang.Runnable: Besitzt lediglich die Methode *run* und wird im Zusammenhang mit der Ausführung von Code durch einen *Thread* seit Java 1.0 verwendet. Wenn eine Klasse *Runnable* implementiert und ein Thread erzeugt wird, so wird der Code der implementierten Methode *run* ausgeführt.
- java.util.concurrent.Callable: Auch dieses Interface wird zur Ausführung von Code in einem eigenen Thread verwendet. Funktioniert sehr ähnlich wie *Runnable*, jedoch kann *Callable* ein Resultat zurückliefern. Wurde in Java 5 hinzugefügt.
- java.util.Comparator: Über dieses Interface kann eine Funktion übergeben werden, mit dessen Hilfe die Reihenfolge einer Kollektion von Daten geordnet werden kann. Wird beispielsweise bei *Collections.sort* eingesetzt und ist seit Java 1.2 Bestandteil der Sprache.
- java.awt.event.ActionListener: Im Zusammenhang mit grafischen Benutzeroberflächen in Java findet der *ActionListener* häufig Gebrauch. Dabei wird dieser zu Elementen der Oberfläche über *addActionListener* hinzugefügt und beim Eintreten

einer Aktion wird dann die übergebene Methode *actionPerformed(ActionEvent e)* aufgerufen. Wurde in Java 1.1 hinzugefügt.

In dem nachfolgenden Beispiel wird ein neuer Thread erstellt. Diesem Thread übergeben wir in Form einer anonymen inneren Klasse eine Implementierung des *Runnable*-Interfaces. Diese innere Klasse hat lediglich die Aufgabe, das Verhalten, welches wir an den Thread übergeben wollen, zu kapseln. In diesem Fall wollen wir dem Thread die Methode *System.out.prinln("Hallo Welt!")* übergeben.

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hallo Welt!");
    }
});
```

Runnable verhält sich hier als klassisches Callback-Interface. Es enthält lediglich eine abstrakte Methode *run()*, welche von uns in der anonymen inneren Klasse implementiert wurde. Die Annotation *@FunctionalInterface* wurde erst mit Java 8 hinzugefügt. Sie wird im Verlauf der Einführung zu Lambda-Ausdrücken genauer erklärt.

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

Im Vergleich dazu würde das Beispiel mit Lambda-Ausdrücken folgendermaßen aussehen. Ohne die Syntax von Lambda-Ausdrücken genauer betrachtet zu haben, lässt sich schon auf den ersten Blick erkennen, dass man durch deren Verwendung einiges an überflüssigem Code einsparen kann.

```
Thread thread = new Thread(
    () -> System.out.println("Hallo Welt!"));
```

Wenn nun die Anwendung von inneren anonymen Klassen bereits so weit in Java verbreitet ist, stellt sich die Frage, warum man sich nicht einfach mit deren Existenz zufrieden gibt. Dazu werden im Design-Dokument *The State of Lambda*[Goe13a] mehrere Gründe genannt, welche gegen den Einsatz von anonymen inneren Klassen sprechen. Wie wir oben bereits anhand des Beispiels einer anonymen inneren Klasse gesehen haben, ist die Syntax relativ umständlich einzusetzen. So benötigen wir für einen Zweck, nämlich die simple Übergabe von Verhalten in Form von Code-als-Daten, bereits mehrere Zeilen

Code, wobei ein Großteil davon Standardcode darstellt. Man könnte mit Hilfe von syntaktischem Zucker die Syntax wesentlich verkürzen und dadurch die Verwendung für den Programmierer deutlich vereinfachen. Des Weiteren würde sich durch die Reduzierung des Codes die Lesbarkeit innerhalb des Programmes erhöhen. In diesem Zusammenhang würde sich auch die Wartbarkeit und Wiederverwendbarkeit des Codes verbessern. Diese Art von Standardcode ist in Java des Öfteren zu beobachten, man denke zum Beispiel an *Getter* und *Setter*-Methoden, welche jedes Mal aufs Neue für änderbare Variablen definiert werden, um einen kontrollierten Zugriff auf diese zu ermöglichen. Weitere Beispiele wären die Methoden *toString*, *equals* und *hashCode*, die für jede erstellte Klasse definiert werden sollten. Um Boilerplate-Code in Java zu reduzieren kann man externe Bibliotheken wie beispielsweise Lombok[Aut] verwenden.

Ein weiterer negativer Punkt von anonymen inneren Klassen ist der Zugriff und die Sichtbarkeit von Variablen, die außerhalb der anonymen inneren Klasse im umliegenden Kontext definiert wurden. Hier treten gerade beim Zugriff auf Variablen auf die umschließende Klasse und bei der Verwendung des this-Schlüsselwortes innerhalb der anonymen inneren Klasse Probleme auf. Ein Vergleich der Sichtbarkeit zwischen anonymen inneren Klassen und Lambda-Ausdrücken, bei denen dies vereinfacht wurde, folgt in einem späteren Unterkapitel dieser Arbeit.

3.3 Lambda Syntax

In diesem Unterkapitel soll die Syntax von Lambda-Ausdrücken anhand von Beispielen dargestellt und erklärt werden. Anschließend wird erläutert, in wie fern sich die oben genannten Nachteile von anonymen inneren Klassen, durch das Hinzufügen von Lambda-Ausdrücken in Java 8 verbessert haben.

Bei Lambda-Ausdrücken handelt es sich um anonyme Methoden. Anonym deshalb, weil ihnen kein Name zugewiesen wird. Man kann einem Lambda-Ausdruck jedoch durch Zuweisung zu einer Variable einen Namen geben. Die Syntax von Lambda-Ausdrücken und deren Aufbau ist jenem von Methoden beziehungsweise Funktionen sehr ähnlich. Es gibt eine Anzahl von Parametern welche dem Lambda-Ausdruck übergeben werden können. Links befinden sich die Parameter und rechts befindet sich der Funktionskörper des Lambda-Ausdrucks. Getrennt werden diese durch einen Pfeil, welcher mit -> dargestellt wird. Das folgende Beispiel zeigt einen simplen Lambda-Ausdruck.[Goe13a][G+14]

```
(int x, int y) -> x + y
```

Dieser Lambda-Ausdruck nimmt zwei Parameter, x und y vom Typ int entgegen. Diese werden im Funktionskörper addiert. Wenden wir diesen Lambda-Ausdrück beispielsweise für die Argumente zwei und drei an, so erhalten wir nach Auswertung der Funktion das Ergebnis fünf zurückgeliefert. Die Anzahl der Parameter auf der linken Seite des Lambda-Ausdrucks kann beliebig gewählt werden. Es können auch Lambda-Ausdrücke ohne Eingangsparameter, welche lediglich einen Wert zurückliefern erzeugt werden.

```
() -> 10
```

Die leere Parameterliste wird durch die Klammern *()* gekennzeichnet. Dieser Lambda-Ausdruck liefert uns bei seiner Auswertung den Wert zehn. Es können zwar beliebig viele Parameter verwendet werden, wenn man jedoch mehrere Dutzend Parameter innerhalb eines Lambda-Ausdrucks benötigt, sollte man seinen Lambda-Ausdruck eventuell restrukturieren. Man wird in den meisten Fällen mit wenigen Parametern auskommen. Im nachfolgenden Beispiel wird ein komplexerer Funktionskörper innerhalb des Lambda-Ausdrucks verwendet.

```
(int x, int y) -> {
    if(x<=y) {
        return y-x;
    } else {
        return x-y;
    }
}
```

Betrachtet man den Funktionskörper genauer, so gibt es hier zwei Arten von Syntax. Einzelne Ausdrücke kommen, wie in den vorangehenden Beispielen, ohne Klammerung aus. Dieser Ausdruck wird ausgewertet und anschließend als Ergebnis der Funktion zurückgegeben. Diese Form von Lambdas wird als Expression-Lambda bezeichnet. Beim Expression-Lambda benötigt man daher keine *return*-Anweisung um den Wert zurückzuliefern. Hingegen wird beim Anweisungs-Lambda ein Code-Block, wie bei einer Methode üblich, abgearbeitet und das Ergebnis wird über *return* zurückgegeben. Ebenso muss bei der Blockvariante, sofern vom Funktionskörper ein Rückgabewert verlangt wird, ein *return*-Statement verwendet werden. Innerhalb des Funktionskörpers eines Lambda-Ausdrucks können auch If-Anweisungen und Schleifen verwendet werden. Es ist also durchaus möglich, auch komplexere Lambda-Ausdrücke zu erzeugen.[G+14]

Die Syntax ist für häufig eingesetzte, einfache Lambda-Ausdrücke ausgelegt worden. Bei den Parametern kann der Typ weggelassen werden. Die Typen der Parameter werden über den Kontext, in dem der Lambda-Ausdruck verwendet wird, über einen Typinferenz-Algorithmus vom Compiler hergeleitet. Des Weiteren können bei einem Parameter die runden Klammern weggelassen werden. Sofern man die Parameter lediglich an die Methode durchreicht, ohne zusätzliche Operationen durchzuführen, kann der Lambda-Ausdruck mit Hilfe von Methoden-Referenz noch weiter verkürzt werden, wie man im folgenden Beispiel sieht. Als einzigen Eingangsparameter haben wir hier einen String, welcher im Funktionskörper an die Methode *System.out.println* weitergeleitet wird und dadurch bei Auswertung des Lambda-Ausdrucks in der Konsole ausgegeben wird.

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

```
//Der Typ kann weggelassen werden, da der Compiler ihn über
//Typinferenz herleiten kann.
(s) -> { System.out.println(s); }

//Da wir nur ein Argument verwenden, können die runden
//Klammern weggelassen werden.
s -> { System.out.println(s); }

//Bei einem einzigen Ausdruck können die geschwungenen Klammern
//ebenfalls weggelassen werden.
s -> System.out.println(s)

//Über Methoden-Referenzen kann dieser Lambda-Ausdruck
//noch weiter vereinfacht werden.
System.out::println
```

Nun wollen wir uns die Unterschiede zwischen Lambda-Ausdrücken und anonymen inneren Klassen genauer anschauen. Dazu sind im folgenden die Probleme von inneren anonymen Klassen kompakt aufgelistet. Anschließend wird erklärt, in wie fern diese Probleme durch Lambda-Ausdrücke gelöst wurden.[Goe13a]

- Syntax von anonymen inneren Klassen benötigt viel überflüssigen Code.
- Die Sichtbarkeit und der Zugriff auf Variablen vom umliegenden Kontext ist in diesem Zusammenhang komplexer als bei Lambda-Ausdrücken. Auch die Verwendung des Schlüsselworts *this* funktioniert anders als gewohnt. Das Schlüsselwort *this* bezieht sich hier nämlich auf die innere Klasse. Will man auf *this* der umliegenden Klasse zugreifen so muss man explizit den Namen der umliegenden Klasse und anschließend *this* angeben: *OuterClass.this*[GJS+15].
- Class-Loading und Instanziierung sind unflexibel. So wird beispielsweise bei der Verwendung einer anonymen inneren Klasse in *Main* neben *Main.class* ein zusätzliches *.class-File Main$1.class* angelegt. Dadurch müssen für jede anonyme innere Klasse neue Dateien erzeugt werden.
- Auf lokale Variablen die nicht als final gekennzeichnet sind, kann innerhalb der inneren Klassen nicht zugegriffen werden.
- Der Kontrollfluss außerhalb der anonymen inneren Klassen kann nicht beeinflusst werden.

Durch die Einführung von Lambda-Ausdrücken zu Java 8 wurden nicht alle der oben gelisteten Probleme beseitigt, da dies eine zu umfangreiche Änderung dargestellt hätte. In der nachfolgenden Auflistung soll nur ein grober Überblick über die Änderungen gegeben werden.[Goe13a]

- Wie man bereits an den Syntax-Beispielen anonymer innerer Klassen im Gegensatz zur Darstellung von Lambda-Ausdrücken erkennen kann, wurde das erste Problem durch das Hinzufügen von Lambda-Ausdrücken beseitigt. Weiter verbessert beziehungsweise abgekürzt wurde die Syntax durch Methoden-Referenzen.
- Scoping wird bei Lambda-Ausdrücken anders gehandhabt als bei anonymen inneren Klassen. Es wird dabei auf ein lokales Scoping gesetzt. Dies bezieht sich auf die Sichtbarkeit und die Verwendung von Variablen außerhalb des Lambda-Ausdrucks. Somit bezieht sich das Schlüsselwort *this* auf die umliegende Klasse und nicht wie in anonymen inneren Klassen auf die innere Klasse.
- Die unflexible Instanziierung und das Laden der anonymen Klassen wird durch eine andere interne Repräsentation der Lambda-Ausdrücke umgangen. Der JVM-Bytecode-Repräsentation ist später noch ein ganzes Kapitel dieser Arbeit gewidmet.
- Lokale Variablen, welche nicht als final gekennzeichnet werden, können nun ebenfalls verwendet werden. Jedoch nur sofern sich der Wert der Variable noch nicht geändert hat. In Java 8 spricht man dabei von *effektiv final*. Geänderte Variablen können nach wie vor nicht im Kontext von Lambda-Ausdrücken beziehungsweise anonymen inneren Klassen verwendet werden.
- Auf die nicht-lokale Abstraktion des Kontrollflusses wird in Java 8 nicht eingegangen. Da man innerhalb des Lambda-Ausdrucks nur finale beziehungsweise effektiv finale Variablen verwenden kann, kann der Kontrollfluss außerhalb des Lambda-Ausdrucks wie bei anonymen inneren Klassen ebenfalls nicht beeinflusst werden.

3.4 Funktionale Interfaces

Auch wenn anonyme innere Klassen relativ viele Nachteile haben, so haben sie zumindest bei der Umsetzung in der virtuellen Maschine von Java (JVM) einen Vorteil. Sie bauen nämlich durch die Verwendung von Interfaces auf die bereits vorhandenen Möglichkeiten der JVM auf. Es werden die Funktionen mit Hilfe von Instanzen eines Callback-Interfaces in Objekten gekapselt und in der JVM repräsentiert. Da Interfaces schon immer einen wichtigen Teil von Java darstellten trifft sich diese interne Darstellung von Code-als-Daten recht gut. Vor Java 8 bekamen diese Callback-Interfaces den Namen SAM-Typen zugewiesen. SAM-Typ steht dabei für *Single Abstract Method* und bezieht sich darauf, dass es innerhalb des Interfaces nur eine abstrakte Methode gibt, welche dann durch den Entwickler implementiert wird. Bei der Umsetzung von Lambda-Ausdrücken wusste man Anfangs nicht genau, welcher Typ einem Lambda-Ausdruck zugewiesen werden sollte. Man hätte hier, ebenso wie bei Haskell, beispielsweise auf Funktionstypen setzen können. Doch diese müssten dann in irgendeiner Weise auch in der JVM repräsentiert werden. Da die JVM keine Funktionstypen unterstützt, hätte man diese als neues Konstrukt innerhalb der JVM einbetten müssen. Ein Funktionstyp für einen Lambda-Ausdruck mit zwei Parametern und Rückgabewert vom Typ *int* würde wie folgt aussehen: (Int,Int) => Int. Der Funktionstyp stellt also die Signatur der Funktion dar.[Buc14]

Man entschied sich dafür, auf Sprachebene die Repräsentation in dieser Art und Weise beizubehalten. Wie wir jedoch noch sehen werden, unterscheidet sich die JVM-Bytecode-Repräsentation zwischen Lambda-Ausdrücken und anonymen inneren Klassen wesentlich voneinander. Um den SAM-Typen einen passenderen Namen für die Verwendung durch Lambda-Ausdrücke zu geben, wurden jene Interfaces, welche seither als Callback-Interfaces verwendet wurden, in Java 8 zu *Funktionalen Interfaces* ernannt. Die Interfaces werden dabei vom Compiler über *@FunctionalInterface* erkannt. Diese dürfen nur eine abstrakte Methode deklarieren. Statische und default-Methoden von Interfaces, die ebenfalls neu in Java 8 zur Sprache hinzugekommen sind, werden dabei nicht dazu gezählt.So wurden nun auch ältere SAM-Typ-Interfaces mit der neuen Annotation *@FunctionalInterface* versehen. Zusätzlich dazu hat man bei Java 8 im Paket *java.util.function* neue funktionale Interfaces hinzugefügt, welche bei der Verwendung von Lambda-Ausdrücken hilfreich sind. Diese werden im nächsten Unterkapitel genauer betrachtet[Orab].[Goe13a]

3.4.1 Neue Pakete in Java 8

Dieses Unterkapitel widmet sich den neu hinzugekommenen Paketen in Java 8. So wurden die beiden Pakete *java.util.function* und *java.util.stream* zu Java hinzugefügt. Des Weiteren wurden auch einige bereits bestehende Pakete modifiziert und erweitert.[G+14]. Nachfolgend werden kurz die Neuerungen, die über *java.util.function* hinzugefügt wurden vorgestellt. Dem anderen Paket *java.util.stream*, auch als *Streams API* bekannt, wird in dieser Arbeit noch ein ganzes Kapitel gewidmet. Durch diese Pakete, vor allem durch die *Streams API*, werden Lambda-Ausdrücke als neues Sprachkonzept aktiv in die Sprache eingebunden. Damit ist gemeint, dass beispielsweise durch den neuen Abstraktionsmechanismus des Streams Programmierer dazu ermutigt werden, Lambda-Ausdrücke vermehrt zu verwenden. Diese beiden Pakete arbeiten sehr eng zusammen. So verwenden relativ viele Methoden von *java.util.stream* die funktionalen Interfaces von *java.util.function* um den verwendeten Lambda-Ausdrücken einen passenden Kontext zu bieten. Die *JSR 335 API Summary*, welche Bestandteil von [G+14] ist, bietet einen sehr guten Überblick über neue und modifizierte Pakete und die dazugehörigen Klassen in Java 8.

3.4.2 java.util.function

Hier werden die in Java 8 neu hinzugekommenen funktionalen Interfaces im Paket *java.util.function* genauer betrachtet. Diese sollen dazu dienen, den Lambda-Ausdrücken und den Methoden-Referenzen einen passenden Kontext für deren Verwendung bereitzustellen. Das Paket beinhaltet dreiundvierzig funktionale Interfaces, wobei die Folgenden als Basis-Interfaces angesehen werden können.[Orab]

- *Predicate<T>*: repräsentiert eine Eigenschaft eines Objektes.
- *Consumer<T>*: Hier wird eine Funktion auf den Eingangsparameter T angewandt. Liefert *void* zurück.

- *Function<T,R>*: Eine Funktion mit Eingangsparameter T, liefert das Resultat vom Typ R.
- *Supplier<T>*: Hat keine Eingangsparameter und liefert eine Instanz vom Typ T zurück.

Bei den anderen bereitgestellten funktionalen Interfaces des Pakets handelt es sich um Spezialisierungen der oben aufgelisteten funktionalen Interfaces. In den nachfolgenden Beispielen sollen die vier funktionalen Interfaces veranschaulicht und deren Einsatzgebiete genauer vorgestellt werden.[Orab]

Predicate

Das funktionale Interface *Predicate* repräsentiert eine Eigenschaft eines Objektes vom generischen Typ *T*. Die abstrakte Methode *test* liefert *true* oder *false* zurück, je nachdem ob das übergebene Objekt die gewünschte Eigenschaft erfüllt. Das bedeutet, wir übergeben dem *Predicate* einen Lambda-Ausdruck, welcher über einen Parameter das zu testende Objekt entgegennimmt und über dessen Funktionskörper ein oder mehrere Vergleichsoperatoren auswertet.

```
Predicate<Person> isAdult = p -> p.getAge() >= 18;
isAdult.test(person);
```

In diesem Beispiel soll über das Prädikat *isAdult* überprüft werden, ob eine Person Erwachsen ist. Der Lambda-Ausdruck, der uns diese Überprüfung ermöglicht, erhält einen Eingangsparameter *p* vom Typ Person. Im Funktionskörper wird über *p.getAge* das Alter der Person abgerufen und über den Vergleichsoperator >= überprüft, ob die Person achtzehn Jahre oder älter ist. Über die Methode *test*, der wir eine Person übergeben, wird die Eigenschaft, welche von *Predicate<Person>* repräsentiert wird, überprüft.

In dem funktionalen Interface *Predicate* befinden sich neben der abstrakten Methode noch drei *Default*-Methoden und eine statische Methode. Die *Default*-Methoden werden für die Komposition von Prädikaten verwendet. Dabei können diese entweder mittels *and* und/oder *or* miteinander verknüpft werden. Über *negate* kann ein Prädikat logisch negiert werden. Durch diese Komposition kann die Lesbarkeit des Codes, aber auch die Wiederverwendbarkeit und Wartbarkeit erhöht werden. Die statische Methode *isEqual* kann dazu verwendet werden, ein Prädikat zu erzeugen, welches auf Objekt-Gleichheit testet.[Orab]

Consumer

Das funktionale Interface *Consumer* repräsentiert eine Funktion mit einem Eingabeparameter vom generischen Typ *T* und liefert kein Resultat zurück. Im Vergleich zu anderen funktionalen Interfaces wird von *Consumer<T>* erwartet, dass hier Seiteneffekte

ausgelöst werden, beispielsweise eine Ausgabe über *System.out.println* oder das Schreiben auf eine externe Datei.[Orab]

```
Consumer<String> printString = s -> System.out.println(s);
printString.accept("String");
```

Die Operation, welche mit Hilfe des Lambda-Ausdrucks für das funktionale Interface *Consumer<String>* festgelegt wird, kann über die Methode *accept* durchgeführt werden. Als Argument nimmt diese Methode das zu konsumierende Objekt, in diesem Beispiel vom Typ *String* entgegen. Komposition wird auch hier durch eine *Default*-Methode *andThen* ermöglicht. Im folgenden Code-Block sieht man, wie *andThen* in Java 8 implementiert wurde. [Orab]

```
default Consumer<T> andThen(Consumer<? super T> after) {
    Objects.requireNonNull(after);
    return (T t) -> { accept(t); after.accept(t); };
}
```

Die beiden aneinandergereihten *Consumer* müssen vom Typ her Kompatibel sein. Wenn der erste *Consumer* den Typ T entgegennimmt, so muss der zweite *Consumer* T oder einen Obertyp von T als Eingangsparameter entgegennehmen.

Supplier

Ein Lambda-Ausdruck, welcher im Kontext des funktionalen Interfaces *Supplier* verwendet wird, besitzt keine Eingangsparameter und liefert ein Resultat vom generischen Typ T zurück. Der Supplier kann daher für die Bereitstellung von Werten oder Objekten verwendet werden, ähnlich wie eine Factory. Über die Methode *get* kann das Resultat des *Suppliers* angefordert werden.

```
Supplier<Double> supplier = () -> Math.random();
supplier.get();
```

So können beispielsweise über den hier festgelegten Versorger über *supplier.get* Zufallszahlen angefordert werden. Dadurch, dass hier Zufallszahlen zurückgegeben werden, handelt es sich jedoch nicht um eine referenziell transparente Funktion.

Function

Das funktionale Interface *Function<T,R>* repräsentiert eine Funktion, welche einen Eingangsparameter vom Typ T annimmt und ein Resultat vom Typ R zurückliefert. Über die Methode *apply* wird die Funktion auf ein Objekt vom Typ T angewandt und

das Resultat dieser Funktion zurückgeliefert. Auch hier gibt es für die Komposition zwei *Default-Methoden*, wobei über *compose* zuerst die zweite Funktion angewandt wird und dessen Ergebnis dann als Eingabeparameter für die erste Funktion verwendet wird. Bei *andThen* wird die Reihenfolge im Vergleich zu *compose* vertauscht. Hat man beispielsweise zwei Funktionen *A* und *B* so wird bei *A.andThen(B)* zuerst über die Funktion *A* ein Ergebnis berechnet und dieses wird dann als Eingangsparameter für die Funktion *B* verwendet. Des Weiteren befindet sich eine statische Methode *identity* im funktionalen Interface *Function*. Diese Methode gibt eine Funktion zurück, die als Resultat ihr Eingangsargument zurückgibt.[Orab]

Zwei weitere erwähnenswerte funktionale Interfaces, die häufiger Verwendung finden, sind *UnaryOperator<T>* und *BinaryOperator<T>*, wobei es sich bei beiden um Spezialisierungen von *Function<T,T>* beziehungsweise *BiFunction<T,T,T>* handelt. Dabei lassen sich Operatoren für ein oder zwei Argumente definieren. Beim *UnaryOperator* wird eine Funktion mit einem Eingangsparameter repräsentiert, wobei Eingangsparameter und Resultat denselben Typen haben. *BinaryOperator* hat die selben Eigenschaften mit dem Unterschied, dass hier zwei Eingangsparameter vom selben Typ verwendet werden. Für die primitiven Typen *int*, *double* und *long* gibt es außerdem für jedes der Basis-Interfaces, also *Supplier*, *Consumer*, *Function* und *Predicate* eine eigene Variante.[Orab]

3.5 Typen von Lambda-Ausdrücken

Wie man bereits bei der Vereinfachung der Syntax gesehen hat, ist es möglich, bei Lambda-Ausdrücken die Typen der Eingangsparameter wegzulassen. Des Weiteren wird auch der Typ des Rückgabewertes nicht explizit festgelegt. Daher stellt sich nun die Frage, woher der Compiler weiß, welche Typen als Eingangsparameter und als Rückgabewert zulässig sind und welche nicht.

Um diese Frage beantworten zu können, werden wir zuerst die Frage klären, welche Art von Objekt durch einen Lambda-Ausdruck repräsentiert wird. Die funktionalen Interfaces, welche den Lambda-Ausdrücken als Schnittstellen dienen, werden nicht explizit im Lambda-Ausdruck angegeben. Daher muss es eine andere Möglichkeit für den Compiler geben, das repräsentierte Objekt und dadurch den Typ eines Lambda-Ausdrucks herauszufinden. Dies geschieht über den Kontext, in dem der jeweilige Lambda-Ausdruck eingesetzt wird. Da der Typ eines Lambda-Ausdrucks nur über den Kontext bestimmt werden kann und Lambda-Ausdrücke in Form von funktionalen Interfaces Verwendung finden, gilt in Java 8 Folgendes: *Lambda-Ausdrücke müssen immer im Kontext eines funktionalen Interfaces vorkommen[Goe13a].* Dadurch kann der Compiler über den Zieltyp des Kontextes die Typ-Inferenz für den jeweiligen Lambda-Ausdruck durchführen. Folgendes Beispiel soll diesen Sachverhalt verdeutlichen.

```
File directory = new File("C:/");
directory.listFiles( (File f) -> f.isFile() );
```

In diesem Beispiel sollen jene Dateien, die kein Verzeichnis darstellen, über die Methode *listFiles* ausgegeben werden. Dieses Beispiel ließe sich über die Methoden-Referenz *File::isFile* noch vereinfachen, da das File lediglich an die Methode *isFile* weitergereicht wird. Nun wollen wir uns anschauen, welchen Typ dieser Lambda-Ausdruck repräsentiert. Dazu schauen wir uns den Kontext des Lambda-Ausdrucks genauer an. In der Klasse *java.io.File* gibt es drei überladene Methoden mit dem Namen *listFiles*. Überladene Methoden haben denselben Namen und unterscheiden sich nur anhand ihrer Signaturen, also durch die Anzahl und Typ der Parameter[GJS+15]. Die Signaturen der Methoden mit dem Namen *listFiles* sehen folgendermaßen aus.

```
public File[] listFiles()
public File[] listFiles(FileFilter filter)
public File[] listFiles(FilenameFilter filter)
```

Da wir der Methode *listFiles* einen Lambda-Ausdruck übergeben, fällt die erste Methode ohne Argumente bereits weg. Bei *FileFilter* und *FilenameFilter* handelt es sich um funktionale Interfaces.

```
@FunctionalInterface
public interface FileFilter {
    boolean accept(File pathname);
}

@FunctionalInterface
public interface FilenameFilter {
    boolean accept(File dir, String name);
}
```

Die funktionalen Interfaces sind hier über die Annotation *@FunctionalInterface*, aber auch anhand einer einzigen abstrakten Methode *accept* erkennbar. Bei unserem Lambda-Ausdruck können wir über den Kontext, nämlich dass *listFiles* entweder einen FileFilter oder aber einen FilenameFilter benötigt, den Typ des Lambda-Ausdruckes feststellen. Es handelt sich also entweder um einen Lambda-Ausdruck vom Typ FileFilter beziehungsweise FilenameFilter. Damit der Compiler sich hier entscheiden kann, werden die Argumente der jeweiligen abstrakten Methode betrachtet. Bei dem Lambda-Ausdruck aus dem Beispiel wird als Parameter lediglich ein *File f* übergeben. Daher kann der Compiler sicherstellen, dass es sich um einen Lambda-Ausdruck vom Typ *FileFilter* handelt, da *FilenameFilter* zwei Parameter entgegen nimmt. Da bei dem funktionalen Interface *FileFilter* die abstrakte Methode *accept* den Typ des Eingangsparameters bereits festlegt, können wir das Beispiel folgendermaßen vereinfachen.

```
File directory = new File("C:/");
directory.listFiles( f -> f.isFile() );
```

Der Compiler kann über das funktionale Interface mittels Typ-Inferenz erkennen, dass es sich bei dem Parameter *f* um einen Parameter vom Typ *File* handelt. Der Rückgabewert des Funktionskörpers des Lambda-Ausdrucks ist über die Methode des funktionalen Interfaces festgelegt. Durch die Bestimmung des Ziel-Typs ist es auch möglich, dass zwei identische Lambda-Ausdrücke unterschiedliche Typen besitzen.

```
Callable<Integer> callable = () -> 1;
Supplier<Integer> supplier = () -> 1;
```

Beide Lambda-Ausdrücke sind identisch. Sie besitzen keine Parameter und liefern beide den Integer eins zurück. Dennoch besitzen sie unterschiedliche Typen, da der erste Lambda-Ausdruck im Kontext von *Callable<Integer>* verwendet wird und der zweite im Kontext von *Supplier<Integer>*. Die Kompatibilität zwischen Lambda-Ausdruck und Ziel-Typ wird in [Goe13a] allgemein folgendermaßen definiert. Ein Lambda-Ausdruck kann dann einem Ziel-Typen T zugeordnet werden, wenn die folgenden Bedingungen eingehalten werden.

- T ist ein funktionales Interface.
- Der Lambda-Ausdruck hat die selbe Anzahl an Parametern wie die Methode vom funktionalen Interface T und die Typen der jeweiligen Parameter sind im Lambda-Ausdruck und in T identisch.
- Jeder Rückgabewert ist kompatibel mit dem Rückgabewert der Methode von T.
- Jede geworfene Ausnahme im Funktionskörper des Lambda-Ausdrucks ist durch die *throws*-Klausel der Methode in T erlaubt.

In Java 8 wird zwischen acht verschiedenen Kontexten mit Ziel-Typen unterschieden. Innerhalb dieser können Lambda-Ausdrücke eingesetzt werden, sofern es sich beim Ziel-Typ um ein funktionales Interface handelt. Im Anschluss an die Auflistung werden die unterschiedlichen Kontexte an Hand von Beispielen erläutert.[Goe13a]

- Variablen-Deklaration,
- Zuweisung,
- Return-Anweisung,
- Initialisierung eines Arrays,
- als Methoden- oder Konstruktor-Argument,
- im Funktionskörper eines Lambda-Ausdrucks selbst (Funktionen höherer Ordnung),

- Innerhalb eines Bedingungsoperators
- und bei Typ-Umwandlungen

Wie wir bereits bei dem Beispiel mit *consumer* und *supplier* gesehen haben, wird dort der Kontext über die Deklaration der Variable bestimmt. Würden wir die Deklaration und die Zuweisung in zwei separate Zeilen aufteilen, so hätten wir in der zweiten Zeile den Kontext und somit den Typ des Lambda-Ausdrucks über die Zuweisung zur Variable und dessen Typ festgelegt.

Beim dritten möglichen Kontext wird der Typ über den festgelegten Return-Typ bestimmt. Da hier bei der Methode *getSupplier* der Rückgabewert den Typ *Supplier<Integer>* haben muss, ist der Lambda-Ausdrück ebenfalls vom Typ *Supplier<Integer>*.

```
public Supplier<Integer> getSupplier() {
    return () -> 20;
}
```

Bei Array-Initialisierungen verhält es sich ähnlich wie bei Zuweisungen, wobei jedoch der Typ über den Typ vom Array abgeleitet wird. In diesem Beispiel werden über das Array drei int-Operatoren für die Addition, Subtraktion und Multiplikation zweier Zahlen bereitgestellt. Jeder der drei Lambda-Ausdrücke in *binOps* hat den Typ des funktionalen Interfaces *IntBinaryOperator*.

```
IntBinaryOperator[] binOps =
    new IntBinaryOperator[]{
        (x,y) -> x+y, (x,y) -> x-y, (x,y) -> x*y
    };
```

Kommt der Lambda-Ausdruck als Methoden- beziehungsweise Konstruktoren-Argument vor, so muss zusätzlich die Auflösung der Überladung von Methoden in Java berücksichtigt werden. Wie man bereits bei dem Beispiel mit *FileFilter* im Zusammenhang mit Ziel-Typen gesehen hat, muss der Compiler sich bei überladenen Methoden für die Ausführung der optimalen Methode entscheiden. Dazu werden Anzahl und Typen der Argumente der Methode beziehungsweise des Konstruktors herangezogen. Des Weiteren wird bei explizit typisierten Lambda-Ausdrücken, das sind jene Lambda-Ausdrücke, bei denen die Parameter explizit Typen angegeben haben, auch der Typ des Rückgabewerts vom Compiler betrachtet. Dieser ist nämlich auf Grund der Argument-Typen herleitbar. Hingegen wird bei implizit typisierten Lambda-Ausdrücken der Funktionskörper des Lambdas vom Compiler ignoriert, somit wird der Rückgabewert und dessen Typ nicht für die Typisierung des Lambda-Ausdruckes verwendet.[G+14]

Wenn der Compiler sich nicht für eine der überladenen Methoden entscheiden kann, so kann man entweder mit Hilfe von Typ-Umwandlung oder expliziten Parameter-Typen

zusätzliche Typinformationen zur Verfügung stellen. Eine weitere Möglichkeit wäre ein Typ-Zeuge, welcher für generische Parameter eingesetzt werden kann. Dies wird im nachfolgenden Beispiel gezeigt.

```
File[] files = myDir.listFiles((File f) ->  f.isFile());

Stream<String> fileNames =
    Stream.of(files)
          .map(f -> f.getAbsolutePath());
```

Bei diesem Beispiel würde der Compiler den Typ folgendermaßen bestimmen. Als Argument für *Stream.of*, welches aus einem oder mehreren Elementen vom generischen Typ T einen Stream desselben Typs erzeugt, wird *files* verwendet. Da es sich dabei um ein Array aus *File*-Objekten handelt, liefert *Stream.of* einen Stream von Objekten des Typs *File* zurück. Als nächstes betrachten wir die Signatur der Funktion *map*.

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

Die generische Funktion *map* erwartet als Eingabeparameter das funktionale Interface *Function<T,R>*. Dieses beschreibt eine Funktion vom Typ T als Eingangsparameter zum Typ *R* als Rückgabewert. Als Rückgabewert liefert *map* einen neuen Stream vom Typ *R*. Über den Eingangs-Stream wird der generische Typ von *T* festgelegt, in diesem Fall entspricht T also *File*. Über den Funktionskörper des Lambda-Ausdrucks kann der Compiler mittels Typinferenz *R* herleiten. Die angewandte Funktion im Funktionskörper des Lambda-Ausdrucks liefer einen String zurück. Daher repräsentiert der Lambda-Ausdruck den Typ *Function<File,String>*. Dadurch liefert die Funktion *map* einen Stream vom Typ *String* zurück. Könnte der Compiler den Typ nicht herleiten, so kann man ihm wie vorher bereits erwähnt, mit einem Typ-Zeugen für den generischen Typ *R* unterstützen. Der Typ-Zeuge *<String>*, der vor der Methode *map* angegeben wird, gibt dem Compiler die nötigen Informationen, um die restlichen Typen herleiten zu können. Man könnte auch direkt die Umwandlung zum funktionalen Interface vornehmen. Darauf sollte jedoch verzichtet werden[Naf14].

```
//Typ-Zeuge für map
Stream.of(files).<String>map(f -> f.getAbsolutePath());

//Umwandlung
Stream.of(files).map(
    (Function<File,String>) f -> f.getAbsolutePath());
```

Lambda-Ausdrücke können auch weitere Lambda-Ausdrücke beinhalten. Es ist also in Java 8 auch möglich Funktionen zu schreiben, die ihrerseits wiederum Funktionen zurückliefern. Solche werden als Funktionen höherer Ordnung bezeichnet.

```
Supplier<IntBinaryOperator> supplier = () -> (x,y) -> x+y;
binaryOperatorSupplier.get().applyAsInt(1,2);
```

Hier liefert der *Supplier* einen *IntBinaryOperator* zurück. Es wird also vom Lambda-Ausdruck eine Funktion, in Form eines Lambda-Ausdrucks zurückgeliefert. Der *Supplier* enthält keine Parameter und liefert die Funktion mit Parametern x und y, dessen Typen über das funktionale Interface *IntBinaryOperator* durch Typ-Inferenz hergeleitet wird. Der Rückgabewert entspricht der Summe der beiden Parameter, ebenfalls vom Typ *int*. Die Typen können also auch hier ohne Probleme durch den Kontext vom Compiler hergeleitet werden. Die zweite Zeile zeigt, wie die Funktion vom *Supplier* mittels *get* geholt wird und über *applyAsInt* auf die Parameter 2 und 3 angewandt wird.

Auch innerhalb eines Bedingungsoperators kann über den Kontext außerhalb des Operators auf die Typen des Lambda-Ausdruckes geschlossen werden. Bei beiden Lambdas handelt es sich in folgendem Beispiel um das selbe funktionale Interface *IntBinaryOperator*. Die Variable *addieren* stellt hier einen booleschen Wert dar. Ist dieser *true*, so wird der erste Lambda-Ausdruck zurückgeliefert, andernfalls wird der zweite zurückgeliefert, welcher die Berechnung der Subtraktion zweier Werte darstellt.

```
IntBinaryOperator addOrSub =
    addieren ? ((x,y) -> x+y) : ((x,y) -> x-y);
```

Über eine Typ-Umwandlung kann ebenfalls ein geeigneter Kontext für einen Lambda-Ausdruck bereitgestellt werden. So wäre diese Zuweisung des Lambda-Ausdrucks zum *Object* im folgenden Beispiel ohne die expliziten Umwandlung zum funktionalen Interface *Callable* nicht möglich. Das Objekt vom Typ *Object* muss vor Verwendung des Lambda-Ausdrucks allerdings nochmals zum Typ *Callable* umgewandelt werden.

```
Object object = (Callable) () -> 10;
```

3.6 Methoden-Referenzen

Wie wir bereits in Beispielen gesehen haben, gibt es in Java 8 eine Möglichkeit gewisse Lambda-Ausdrücke durch Methoden-Referenzen zu vereinfachen. Anstelle einer namenlosen Methode, die mittels Lambda-Ausdruck erzeugt wird, kann man so auf bereits vorhandene Methoden zugreifen und diese übergeben. Dadurch kann bei dem folgenden Beispiel, in dem der String s lediglich an die Methode *println* von *System.out* weitergegeben wird, durch die Methoden-Referenz *System.out::println* ersetzt werden.[Goe13a]

```
s -> System.out.println(s)
System.out::println
```

Man kann sich die Methoden-Referenz dabei wie einen Lambda-Ausdruck vorstellen, letztendlich verhalten sie sich auch wie solche. Ebenso wie ein Lambda-Ausdruck darf die Methoden-Referenz nur im Kontext eines Ziel-Typs vorkommen und muss eine Instanz eines funktionalen Interfaces darstellen. So könnte die Methoden-Referenz *System.out::println* beispielsweise im Kontext des funktionalen Interfaces *Consumer<String>* vorkommen. Der Vorteil liegt an der teilweise kürzeren Syntax, und des Weiteren kann man klarer vermitteln, dass es sich hierbei um eine Methode handelt, die bereits einen Namen besitzt, also schon an einer anderen Stelle definiert wurde. In [Goe13a] wird zwischen sechs verschiedenen Arten von Methoden-Referenzen unterschieden.

- Referenzierung auf eine statische Methode. (*KlassenName::methodenName*)
- Referenzierung auf eine Instanz-Methode eines bestimmten Objektes. (*klassenInstanz::methodenName*)
- Referenzierung auf eine *super*-Methode eines Objektes. Hier wird also auf eine Methode des Obertyps des Objektes verwiesen. (*super::methodenName*)
- Referenzierung auf eine Instanz-Methode eines beliebigen Objektes eines bestimmten Typs. (*KlassenNameDesTyps::methodenName*)
- Eine Referenz auf einen Konstruktor einer Klasse. (*KlassenName::new*)
- Eine Referenz auf einen Konstruktor eines Arrays. (*TypName[]::new*)

Bei einer statischen Methode wird lediglich der Name der Klasse mit beginnendem Großbuchstaben, gefolgt von einem :: und dem jeweiligen Methodennamen für die Methoden-Referenz angegeben. Zum Beispiel: *String::join*.

Bei der Referenzierung auf eine Methode eines bestimmten Objektes wird vorher eine Referenz auf das Objekt benötigt. Ansonsten verhält es sich wie bei der statischen Methode, nur wird statt der Klasse die Objekt-Referenz angegeben. Mit Hilfe der Methoden-Referenz kann man Lambda-Ausdrücke auch relativ einfach zwischen den einzelnen funktionalen Interfaces umwandeln. Im folgenden Beispiel wird ein Lambda-Ausdruck vom Typ *Supplier* über die Methoden-Referenz zu einem Lambda-Ausdruck vom Typ *Callable*.

```
Supplier<Integer> supplier = () -> 1;
Callable<Integer> callable = supplier::get;
```

Im folgenden Beispiel wird die Methoden-Referenz auf ein beliebiges Objekt eines bestimmten Typs angewandt. Hier soll das Argument *s* vom Typ *String* mit Hilfe der Methode *toLowerCase* in Kleinbuchstaben umgewandelt werden. Bei dieser Art von Methoden-Referenz wird der Typ des Arguments vor dem Trennzeichen *::* und der jeweiligen Methode, welche auf das beliebige Objekt des Typs angewandt werden soll, hingeschrieben.

```
Function<String, String> toLowerCase =
    (String s) -> s.toLowerCase();
Function<String, String> toLowerCase = String::toLowerCase;
```

Die Syntax einer Methoden-Referenz auf Instanzen einer Klasse gleicht jener von statischen Methoden von Klassen. Es beginnen beide mit dem Klassennamen in Großbuchstaben, gefolgt durch das Trennzeichen :: und gefolgt vom Namen der Methode. Daher muss der Compiler in solchen Fällen entscheiden, um welche Methode es sich handelt.[Goe13a]

3.7 Interface-Evolution

Das Hinzufügen des Sprachkonzepts der Lambda-Ausdrücke zu Java 8 bringt, wie wir bereits gesehen haben, einige Vorteile mit sich. Doch um das Potenzial der funktionalen Programmierung besser ausschöpfen zu können, müssen sich auch Bibliotheken an dieses neue Konzept anpassen. Deshalb wurden in Java 8 neben den bereits erwähnten Lambda-Ausdrücken, Methoden-Referenzen und funktionalen Interfaces, zwei weitere Konzepte hinzugefügt: Default-Methoden und statische Methoden in Interfaces.

Java hatte vor Version 8 folgendes Problem im Bezug auf Interfaces und die Weiterentwicklung dieser. Sobald ein Interface zu Java hinzugefügt wurde, hatte man praktisch kaum mehr eine Möglichkeit, neue Methoden nachträglich zu diesem Interface hinzuzufügen. Alle Methoden die in einem Interface vor Java 8 vorkamen, mussten nämlich in der Klasse, die das Interface implementiert, entweder von einem Obertyp eine Implementierung geerbt haben oder aber selber die jeweilige Methode implementieren. Hätte man nun bei einer neuen Version von Java neue Methoden zu einem Interface hinzugefügt, so hätte man die neuen Methoden ebenfalls in allen Implementierungen des Interfaces hinzufügen müssen. Somit hätte man zusätzlich zu dem Interface auch alle dazugehörigen Implementierungen umschreiben müssen. Da gerade die Standardbibliotheken von Java, wie beispielsweise die *Collection API* von vielen Personen verwendet wird, war es für das Team rund um Java 8 nicht möglich, ohne neue Sprachkonstrukte hinzuzufügen, die Bibliotheken für den Einsatz von Lambda-Ausdrücken aufzurüsten. Dieses Problem bestand generell für die Weiterentwicklung von Bibliotheken und deren Interfaces, bezieht sich also nicht nur auf die Standardbibliotheken von Java.[Goe13b]

Eine andere Möglichkeit um neue Methoden hinzuzufügen wäre es, diese als statische Methoden festzulegen. Da vor Java 8 jedoch keine statischen Methoden in Interfaces möglich waren, wurden diese meist in eine eigene Klasse für Hilfsmethoden hinzugefügt. So existiert bei den häufig verwendeten Interfaces, wie zum Beispiel der *Collection API* eine Klasse für statische Methoden, wobei als Name die Mehrzahl des Interface-Namens verwendet wird. In diesem Fall könnte man statische Methoden bei der Klasse Collections.java hinzufügen. Dort befinden sich aber nur Hilfsmethoden. Das Hinzufügen würde die neuen Methoden für die Verwendung von Lambda-Ausdrücken zu Methoden zweiter Klasse abstempeln[Oraa]. Ebenso wollte man auch keine *Collection II* API

herausbringen. Um die genannten Probleme zu umgehen wurden in Java 8 *Default*-Methoden und statische Methoden für Interfaces hinzugefügt. Durch *Default*-Methoden lassen sich Interfaces ab Java 8 problemlos weiterentwickeln. Über die statischen Methoden in Interfaces können zukünftig Hilfsklassen, in denen lediglich statische Methoden definiert sind, eingespart werden.[Goe13a]

Hier als Beispiel für die Syntax einer *Default*-Methode die neu hinzugefügte Methode *sort* zum *List*-Interface.

```
public interface List<E> extends Collection<E> {
    default void sort(Comparator<? super E> c) {
        Collections.sort(this, c);
    }
```

Nachdem nun auch in Interfaces die Möglichkeit besteht Methoden zu implementieren, stellt sich die Frage, in wie fern sich in Java 8 Interfaces und abstrakte Klassen voneinander unterscheiden. Eine Klasse kann nach wie vor nur eine abstrakte Klasse erweitern, beziehungsweise von dieser erben. Bei Interfaces gibt es diese Beschränkung nicht, wie man bei dem Beispiel im Unterkapitel *Vererbung von Default-Methoden* sehen kann. Ein weiterer wesentlicher Unterschied ist, dass abstrakte Klassen Instanz-Variablen festlegen können und dadurch einen Zustand repräsentieren können. Auf diese Variablen kann dann auch durch eine erweiterte Klasse zugegriffen werden. Im Gegensatz dazu sind bei Interfaces alle Variablen automatisch als *static* und *final* deklariert.[UFM14]

3.7.1 Einsatzmöglichkeiten von Default-Methoden

Neben dem bereits erwähnten Einsatzszenario bei der Weiterentwicklung von Bibliotheken und APIs werden in [UFM14] noch zwei weitere interessante Möglichkeiten vorgestellt. So wird es unter anderem auch in der Java API innerhalb des Interfaces *Iterator* verwendet um optionale Methoden zu repräsentieren. Das Interface *Iterator* wird unter anderem vom *Collection*-Interface implementiert um beispielsweise eine Liste von Elementen zu durchlaufen. Die Operation *remove* wurde meistens bei der Implementierung von *Iterator* weggelassen. Die Programmierer haben hier meist eine *UnsupportedOperationException* geworfen. Dadurch, dass diese Methode dennoch immer implementiert werden musste, auch wenn die Implementierung dieser lediglich eine Exception geworfen hat, fiel jedes mal zusätzlicher Code an. In Java 8 wurde *remove* deshalb mit Hilfe von *default*-Methoden zu einer so genannten optionalen Methode.

```
interface Iterator<T>{
    boolean hasNext();
    T next();
    default void remove() {
        throw new UnsupportedOperationException();
    }
```

```
}
```

Eine weitere Möglichkeit ist die Mehrfach-Vererbung. Wie bereits erwähnt kann in Java eine Klasse nur von einer konkreten oder abstrakten Klasse erben, jedoch mehrere Interfaces implementieren. Daher gab es schon vor Java 8 Mehrfach-Vererbung in Hinsicht auf Typen beziehungsweise Deklaration. Nun kommt in Java 8 durch *Default*-Methoden die Mehrfach-Vererbung von Verhalten dazu. So kann eine Klasse über mehrere Interfaces Implementierungen von Methoden übernehmen. Um die Komplexität einer Klasse nicht unnötig zu erhöhen, sollte man mit dieser Art der Vererbung jedoch vorsichtig umgehen. Es ist nicht sinnvoll, von einem Interface mit beispielsweise hunderten Methoden zu erben nur um eine Methode davon auszuführen. Im Zusammenhang mit Mehrfach-Vererbung denkt man meistens an das *Diamond*-Problem in *C++*. Dabei geht es darum, dass eine Klasse jeweils zwei Klassen erweitert, die wiederum eine gemeinsame Klasse erweitern. In *C++* kann eine Klasse jedoch von mehreren Klassen erben, daher ist es auch eine Mehrfach-Vererbung in Hinsicht auf den Zustand einer Klasse. Dort treten dann durch die Speicher-Referenzen auf die jeweiligen Oberklassen Probleme auf. Im nachfolgenden Abschnitt *Vererbung von Default-Methoden* sieht man wie dieses Problem in Java gelöst wird; in Hinsicht auf Mehrfach-Vererbung von Verhalten.[UFM14]

3.7.2 Vererbung von Default-Methoden

Grundsätzlich verhält es sich bei der Vererbung von Default-Methoden wie bei der herkömmlichen Vererbung von Methoden einer Klasse oder eines Interfaces. Es gibt jedoch Ausnahmen. Wenn beispielsweise von mehreren Obertypen einer Klasse oder eines Interfaces dieselben Methoden mit gleicher Signatur bereitgestellt werden, gibt es zwei Regeln über die der Compiler entscheidet, welche davon an den Untertypen vererbt wird.[Goe13a]

- Methoden von konkreten und abstrakten Klassen werden im Gegensatz zu den *Default*-Methoden bevorzugt. Also nur wenn über die Klassen-Hierarchie keine passende Methode festgelegt wird, wird die jeweilige *Default*-Methode verwendet.
- Methoden die bereits von einem anderen Kandidaten überschrieben worden sind, werden vom Compiler ignoriert. Das bedeutet, je spezieller das Interface umso eine höhere Priorität hat es bei der Auswahl der Methode durch den Compiler.

In diesem Beispiel soll die erste Regel bei der Vererbung von *default*-Methoden verdeutlicht werden.

```
public interface Interface {
    default void test() {
        System.out.println("Interface");
    }
```

```
}

public abstract class AbstractClass {
    public void test() {
        System.out.println("Abstrakt");
    }
}

public class Class extends AbstractClass implements Interface {}
```

Die Klasse erweitert hier die abstrakte Klasse und implementiert das Interface. Beide beinhalten die Methode *test* mit gleicher Signatur. Durch die erste Regel wird die Methode der abstrakten Klasse bevorzugt. Daher würde das Aufrufen der Methode *test* in *Class* den String *Abstrakt* zurückliefern.

Wenn man in der abstrakten Klasse die Methode *test* ohne Implementierung angeben würde (*abstract void test()*), so müsste man in der abgeleiteten Klasse eine Implementierung der Methode angeben, obwohl eine *default*-Methode vom Interface bereitgestellt wird. Will man die *Default*-Methode des Interfaces verwenden, so muss die überschriebene Implementierung der Methode in der Klasse mit *Interface.super.test()* die *Default*-Methode aufrufen. Im nächsten Beispiel soll die Verwendung der zweiten Regel durch den Compiler geschildert werden.

```
interface A {
    default void test() {
        System.out.println("Interface A");
    }
}

public interface B extends A {
    default void test() {
        System.out.println("Interface B");
    }
}

public interface C extends A {}

public class Class implements InterfaceB, InterfaceC {}
```

Hier implementieren die Interfaces B und C das Interface A, wobei B die Methode *test* von A überschreibt und C die Methode *test* von A vererbt bekommt. Die Klasse implementiert die beiden Interfaces B und C. Welcher String wird nun ausgegeben wenn man innerhalb der Klasse die Methode *test* aufruft? Der Compiler kann über die zweite

Regel feststellen, welche Methode aufgerufen werden muss. Die zweite Regel besagt, dass Methoden, welche bereits in einem anderen abhängigen Interface überschrieben wurden, vom Compiler ignoriert werden. Da *B* bereits die Methode *test* von *A* überschrieben hat, wird die geerbte Methode in *C* vom Compiler ignoriert. Der Aufruf der Methode in der implementierten Klasse liefert den String *Interface B* zurück.

Es gibt noch einen weiteren Fall, wobei der Compiler hier nicht über die zwei oben beschriebenen Regeln zur Ableitung von *default*-Methoden über den korrekten Methodenaufruf bestimmen kann. Im folgenden Beispiel wird dies genauer beschrieben.

```
public interface A {
    default void test() {
        System.out.println("Interface A");
    }
}

public interface B {
    default void test() {
        System.out.println("Interface B");
    }
}

public class Class implements A,B {
    @Override
    public void test() {
        InterfaceA.super.test();
    }
}
```

Wenn eine Klasse zwei voneinander unabhängige Interfaces implementiert, wobei beide Interfaces mindestens eine Methode mit gleicher Signatur beinhalten, so kann der Compiler nicht über die Auswahl der vererbten Methode entscheiden und wirft eine Fehlermeldung. In diesem Beispiel implementiert die Klasse zwei Interfaces *A* und *B*. Diese stehen in keiner Relation zueinander, implementieren also nicht ein und dasselbe Interface als Obertypen. Beide Interfaces haben eine *default*-Methode für *test*, wobei es sich um Methoden mit gleicher Signatur handelt. Der Compiler liefert hier die folgende Fehlermeldung.

```
Class inherits unrelated defaults for test() from A and B
```

Da sich der Compiler in diesem Fall nicht für eine der beiden *default*-Methoden entscheiden kann, muss die Entscheidung programmatisch getroffen werden. Dazu wurde in Java 8 eine neue Art, Obertypen zu referenzieren, hinzugefügt. In dem Beispiel wird von der Klasse mit Hilfe der neuen Syntax *A.super.test()* die *default*-Methode von *A* übernommen.

Vor Java 8 war die Referenzierung auf Obertypen nur durch *super* möglich. Wenn die Interfaces ihrerseits jedoch auch ein Interface implementieren, so ist der Zugriff auf Methoden dieses Interfaces nicht über *InterfaceName.super* möglich. Es können immer nur direkte Super-Interfaces angesprochen werden.

Wenn zwei Interfaces eine Methode mit gleicher Signatur deklarieren, wobei ein Interface die Methode als *default* deklariert und das andere Interface als abstrakte Methode, verhält es sich genauso wie in dem vorherigen Beispiel mit dem Interface und der abstrakten Klasse. Es muss also eine Implementierung der Methode in der Klasse, welche beide Interfaces implementiert, vorkommen.[Goe13b]

Auf Grund der hier definierten Regeln kann es in Java zu keinem *Diamond*-Problem kommen, da entweder Regeln für den Compiler aufgestellt sind, für welche Methode er sich zu entscheiden hat, oder man muss manuell über *InterfaceName.super.methodenName* angeben, welche Methode die bevorzugte ist.

3.7.3 Statische Methoden in Interfaces

Mit Hilfe von statischen Methoden kann man bei zukünftigen Interfaces die Klasse mit den Hilfsmethoden weglassen und die statischen Methoden direkt zum Interface hinzufügen. Bei der *Collection API* in Java 8 wurde noch auf statische Methoden verzichtet. Die statischen Hilfsmethoden befinden sich nach wie vor in der Klasse *Collections*. Bei der neu hinzugekommenen Klasse *Stream*, welche im nächsten Kapitel vorgestellt wird, werden die statischen Methoden bereits im Interface definiert. Es gibt zwar die Klasse *StreamSupport*, in der Hilfsmethoden festgelegt werden, jedoch sind diese eher für das Programmieren von Bibliotheken interessant.[Orab]

KAPITEL 4

Streams API

Im vorherigen Kapitel dieser Arbeit wurden bereits die zwei neuen Pakete *java.util.function* und *java.util.stream* erwähnt. Dabei wurde das Paket *java.util.function* mit dessen funktionalen Interfaces bereits genauer vorgestellt. Diese werden häufig von den Java Standardbibliotheken eingesetzt und bieten Abstraktionen für Lambda-Ausdrücke, wie beispielsweise eine Funktion, einen Konsumenten oder einen Lieferanten von Werten. Sie stellen also Basiskonstrukte für den Einsatz von Lambda-Ausdrücken bereit. Um den Einsatz von funktionaler Programmierung in Java weiter voranzutreiben, wurde nun auch die *Collections API* erweitert. Diese weit verbreitete API war noch nicht für den Einsatz von Lambda-Ausdrücken bereit. Anfangs hatte man hier überlegt, die seit Java 1.2 bestehende Bibliothek komplett zu überarbeiten und als *Collections II* API anzubieten. Man entschied sich dann aber gegen diesen Vorschlag, da die *Collections API* in Java in den unterschiedlichsten Richtungen einen hohen Grad an Verwendung findet. Dadurch wäre nicht nur die komplette Überarbeitung dieser ein enormer Aufwand gewesen, sondern es hätte auch einige Jahre gedauert, bis die Änderungen auf allen abhängigen Implementierungen abgeschlossen gewesen wären. Eine komplette Überarbeitung der *Collections API* in kommenden Java-Versionen wird dennoch nicht ausgeschlossen.[Goe13b]

Stattdessen wurde nun ein neuer Abstraktionsmechanismus, nämlich der *Stream* hinzugefügt. Die bereits bestehenden Bibliotheken, welche durch den Einsatz von *Streams* profitieren können, wurden um passende Methoden zur Erzeugung dieser erweitert. So wurden der *Collections-API*, aber auch anderen Klassen, wie beispielsweise *java.util.Random* oder *Files*, welche statische Methoden für die Verarbeitung von Dateien bereitstellt, Methoden hinzugefügt, welche die jeweiligen Objekte in einen *Stream* umwandeln. Die Operationen, die auf Streams angewandt werden können, nehmen als Parameter meist einen Lambda-Ausdruck vom Typ eines funktionalen Interface, relativ häufig aus dem bereits vorgestellten Paket *java.util.function*, entgegen.[Goe13b]

Im diesem Kapitel wird zuerst kurz erklärt, worum es sich bei einem Stream handelt. Anschließend werden die Eigenschaften von Streams erläutert. Dadurch werden die

Unterschiede im Gegensatz zu Kollektionen der *Collections-API* verdeutlicht. Danach wird die interne mit der externen Iteration im Zusammenhang mit dem Durchlaufen einer Sequenz von Elementen vorgestellt. Dann wird erklärt, wie sich in Java 8 Streams erzeugen lassen und welche Operationen auf die Elemente der Streams angewandt werden können. Die Erzeugung von Streams und die Anwendung der Operationen wird dabei durch kurze Beispiele erläutert.

Abschließend wird der interne Aufbau der relevanten Stream-Klassen vorgestellt, also wie Streams in Java aufgebaut sind. In diesem Zusammenhang wird auch kurz auf das Fork-Join-Framework, welches bereits in Java 7 hinzugefügt wurde, eingegangen. Das Fork-Join-Framework ist deshalb relevant weil es im Zusammenhang mit parallelen Streams verwendet wird. Am Ende des Kapitels sind noch einige Vor- und Nachteile im Vergleich zur Verwendung von Schleifen angeführt.

4.1 Streams: Einführung

Bei einem Stream handelt es sich um eine Sequenz von Objekt-Referenzen. Mit Hilfe von Stream-Operationen wird es ermöglicht, diese Sequenz einfach zu verarbeiten. Die Objekte können dabei mit Hilfe von Lambda-Ausdrücken in einer deklarativen Art und Weise verarbeitet werden. Im Gegensatz zur *Collections API* wird bei der *Streams API* nur übergeben, was gemacht werden soll, und nicht, wie es gemacht werden soll. Der Programmierer legt fest, was gemacht werden soll und überlasst die genaue Durchführung der jeweiligen Bibliothek. In diesem Zusammenhang unterscheidet man zwischen interner und externer Iteration der Datensätze. Die deklarative Art, aber auch die vereinfachte Möglichkeit, Stream-Operationen parallel abzuarbeiten, beruht darauf, dass Streams wie eine Pipeline von Operationen aufgebaut sind. Da ein Teil der Operationen auf Streams selbst wieder Streams zurückliefert, können die Operationen mit einem Punkt aneinandergehängt werden. Dies wird als Method-Chaining, also als Verkettung der Methoden bezeichnet. Eine andere Bezeichnung dafür wäre das so genannte *Fluent Interface*. Aufgrund dieser Verkettung der Operationen liest sich der Code, den man mit der Streams API zur Verarbeitung der Objekte schreibt, ähnlich wie die Problemstellung, die in diesem Zusammenhang gelöst werden soll[Buc14]. Im Vergleich dazu musste man vor Java 7 bei Verwendung der *Collections API* meist mehrere, teilweise verschachtelte for-Schleifen und if-Abfragen verwenden, um mehrere Operationen auf die Objekte der Collection durchzuführen.

Man könnte Streams vereinfacht gesehen mit einer Menge von Daten vergleichen, die mit Hilfe eines Iterators, beziehungsweise über eine *for*-Schleife durchlaufen werden, und auf deren Objekte innerhalb der Schleife Operationen durchgeführt werden. Es gibt jedoch bedeutende Unterschiede zwischen *Collections* und *Streams*, welche im Anschluss näher erläutert werden. Die Objekte des Streams können über die vordefinierten Operationen beispielsweise nach Eigenschaften gefiltert werden, auf andere Datentypen umgewandelt werden und über Aggregat-Funktionen anhand von Eigenschaften zusammengefasst werden. Des Weiteren kann man beispielsweise nach einem Maximum oder Minimum

einer Sequenz von Objekten suchen.

Die größte Motivation beim Hinzufügen dieses neuen Sprachkonzepts war es, einerseits die hinzugefügten Lambda-Ausdrücke besser einsetzbar zu machen und dadurch andererseits die parallele Programmierung einfacher für Entwickler zu ermöglichen. Des Weiteren wird durch die Einführung der *Streams* zusammen mit den Operationen, die auf die Objekte dieser angewandt werden können, eine lesbare, gut wartbare und wiederverwendbare Möglichkeit geboten, große Datenmengen effizient zu verarbeiten. Außerdem sind die Operationen so aufgebaut, dass sie mit Hilfe von Komposition einfach zusammengesetzt werden können. Wie wir sehen werden, ist es ohne Probleme möglich, Streams entweder sequentiell oder durch eine geringe Änderung im Programmcode parallel einsetzen zu können. Der Einsatz von Parallelität soll dabei vereinfacht werden, jedoch nicht komplett transparent vonstattengehen. Daher muss explizit die Verwendung von parallelen Streams bei der Entwicklung angegeben werden. Das Ziel war es hier, die parallele Verarbeitung von Daten einfacher, aber nicht komplett unsichtbar für den Entwickler zu machen. Der Entwickler soll sich somit immer damit auseinandersetzen, ob eine sequenzielle oder parallele Verarbeitung sinnvoller ist. Nicht immer ist eine parallele Abarbeitung auf Grund des Overheads bei der Aufteilung der Tasks schneller als eine sequentielle Abarbeitung der Daten.[Goe13b]

Streams werden in Java 8 über das Interface *Stream* bereitgestellt, wobei es sich dabei um eine Sequenz von Objekten vom generischen Typ T handelt. Zusätzlich dazu gibt es für primitive Datentypen die spezialisierten *IntStream*, *LongStream* und *DoubleStream*. Laut [UFM14] ist die Verwendung von primitiven Datentypen anzustreben, da dadurch ein Performance-Vorteil erzielt werden kann. Diese Aussage werden wir im Zusammenhang mit dem in einem späteren Kapitel durchgeführten Benchmark überprüfen.

4.2 Eigenschaften von Streams

Über die hier genannten Eigenschaften von Streams soll die Abgrenzung und die Unterscheidung im Bezug auf die *Collections API* verdeutlicht werden. Es handelt sich um zwei unabhängige Bibliotheken. Es werden aber Methoden bereitgestellt, um zwischen den Bibliotheken umzuwandeln. Geht es bei den *Collections* eher darum, die Elemente zu Speichern und den Zugriff auf diese komfortabel bereitzustellen, so stellen *Streams* nicht direkt den Zugriff auf einzelne Elemente bereit. Vielmehr geht es bei den Streams um die Verarbeitung von Datensätzen, die über eine Aneinanderkettung von Operationen realisiert wird.[Goe13b]

Im Gegensatz zu Collections werden Streams nicht zum Speichern von Objekten verwendet. Man kann sich einen Stream daher ähnlich wie ein Fließband oder eine Pipeline vorstellen. Am Anfang des Fließbandes werden die Objekte von einer bestimmten Quelle auf das Fließband gelegt. Anschließend werden meist mehrere, beliebig viele Arbeitsschritte durchgeführt. Diese sind mit den Operationen, welche auf den Strom der Daten angewandt werden, vergleichbar. Am Ende des Fließbandes bekommt man das fertige Produkt zurückgeliefert, bei Streams das gewünschte Resultat. Streams und die dazugehörigen

Operationen sind von Grund auf funktional aufgebaut. Operationen liefern das gewünschte Resultat, jedoch ohne die Datenquelle zu beeinträchtigen beziehungsweise zu verändern. Des Weiteren werden bei relativ vielen Operationen auf Streams Lambda-Ausdrücke als Argumente entgegen genommen. Daher handelt es sich bei den meisten Operationen auch um Funktionen höherer Ordnung. In diesem Zusammenhang gibt es auch ein Entwurfsmuster für parallele Verarbeitung von Datensätzen, welches als *Pipeline-Pattern* bekannt ist.[pip]

Streams verwenden eine verzögerte Auswertung, im Englischen als *lazy evaluation* bezeichnet. Ein Teil der verwendeten Operationen, nämlich jene, die einen weiteren *Stream* zurückliefern, verwenden solch eine Art der Auswertung. Dabei wird die Operation nur dann ausgeführt, wenn ein Bedarf an dem Resultat dieser Operation besteht. Die Auswertung der Operation wird verzögert. Wenn man beispielsweise über die Methode *filter* in einem Stream von Objekten des Typs *String* nur das erste Wort, das länger als fünf Buchstaben ist, als Resultat haben möchte, passiert bei der Auswertung Folgendes. Wenn zum Beispiel das erste Wort gleich ein passendes Wort darstellt, wäre es eine Verschwendung von Rechenleistung, die restlichen Strings auch noch durchzugehen. Daher kann hier das Filtern der Strings frühzeitig abgeschlossen werden. Diese Eigenschaft von Streams ermöglicht uns im Gegensatz zu Kollektionen die Verwendung von unendlich großen Streams von Objekten. So könnte man sich die ersten hundert Primzahlen über einen *IntStream* und die *filter*-Methode zurückliefern lassen, wobei der Input des Streams alle natürlichen Zahlen sind. Mit Hilfe von Operationen, welche die Berechnung beziehungsweise den Stream terminieren, ist es möglich, Berechnungen über unendliche Streams in endlicher Zeit mit einem Resultat abschließen zu können.

Streams können nur einmal Durchlaufen werden. Hier verhält es sich wie bei einem Iterator. Sobald einmal eine terminierende Operation auf den Stream angewandt wurde, muss ein neuer Stream angefordert werden. Man kann denselben Stream anschließend nicht nocheinmal für Operationen verwenden. Versucht man dies trotzdem, so wirft der Compiler eine Fehlermeldung. Es wird jedoch in den Java-Docs erwähnt, dass es nicht immer möglich ist, das mehrmalige Verwenden eines Streams, abhängig von dessen Implementierung, festzustellen.[Orab]

4.3 Interne und externe Iteration

Beim Durchlaufen der Elemente einer Folge wird zwischen interner und externer Iteration unterschieden. Vor Java 8 hätte man die Iteration über eine Folge von Elementen wahrscheinlich über eine *for-each*-Schleife durchgeführt. Die *for-each*-Schleife wurde in Java 5 hinzugefügt[Ora04]. Eine weitere Möglichkeit wäre es gewesen, eine herkömmliche *for*-Schleife von 0 bis zur Anzahl der Objekte in der Liste zu durchlaufen und innerhalb der Schleife mit *get(index)* auf das jeweilige Objekt der Liste zuzugreifen. Im folgenden Beispiel ist die externe Iteration mit Hilfe der *for-each*-Schleife dargestellt.[Naf14]

```
for(Person p : personList) {
    System.out.println(p.getName() + "," + p.getAge());
}
```

Hier wird eine Liste von Personen über die *for-each*-Schleife iteriert. Es wird von jeder Person innerhalb dieser Liste der Name und das Alter ausgegeben. Bei der *for-each*-Schleife handelt es sich um syntaktischen Zucker. Er verkürzt, beziehungsweise vereinfacht die Schreibweise dieses Codes. Im Hintergrund wird dafür ein externer Iterator verwendet. So würde die *entzuckerte* Variante des oben stehenden Codes folgendermaßen aussehen.

```
Iterator<Person> iterator = personList.iterator();
while(iterator.hasNext()) {
    Person p = iterator.next();
    System.out.println(p.getName() + "," + p.getAge());
}
```

Es wird explizit über die Methode *iterator* ein Iterator für die Liste der Personen angefordert. Anschließend wird in einer *while*-Schleife mit *hasNext* überprüft, ob die Liste noch ein weiteres Element vom Typ *Person* beinhaltet. Im Schleifenkörper wird dann über *next()* das nächste Element, also die nächste Person aus der Liste abgerufen und der Variable *p* zugewiesen. Danach erfolgt die Ausgabe des Namens und des Alters der Person. Die hier verwendete Iteration wird als externe Iteration bezeichnet. Die Iteration findet dabei außerhalb der jeweiligen Bibliothek statt, nämlich direkt in dem vom Entwickler geschriebenen Code. In diesem Beispiel wird über die Elemente einer Liste iteriert, dies geschieht jedoch außerhalb der Klasse *ArrayList*. Die Programmierer geben nicht nur an, *Was* gemacht werden soll (Ausgabe des Namens und des Alters über *System.out.println)*, sondern auch *Wie* es gemacht werden soll, nämlich sequenziell in einer vorgegebenen Reihenfolge über einen Iterator. Das hat den Nachteil, dass die Bibliothek selbst nichts an der Ausführung mitbestimmen kann. Dadurch wird die *for-each*-Schleife beispielsweise immer sequenziell und in korrekter Reihenfolge durchgeführt, auch wenn in einigen Fällen eine parallele, ungeordnete Reihenfolge der Auswertung effizienter verlaufen würde. Deshalb wird in Java 8 auf die interne Iteration gesetzt. Das obige Beispiel könnte man nun wie folgt umsetzen.

```
personList.forEach(
    p -> System.out.println(p.getName() + " " + p.getAge()));
```

Dabei wird der in Java 8 neu hinzugekommenen Methode *forEach* ein *Consumer* vom bereits vorgestellten *java.util.function* übergeben. Der Code erfüllt die selbe Aufgabe wie die oberen beiden Beispiele, jedoch wird die Iteration der Bibliothek überlassen. Dadurch kann diese über das *Wie* entscheiden, und die Programmierer geben in Form des Lambda-Ausdrucks nur mehr an, was gemacht werden soll.

Streams verwenden ebenfalls diese Art der Iteration, da die Operatoren auf Streams als Eingabeparameter ebenfalls Lambda-Ausdrücke verwenden wie die hier vorgestellte *for-each*-Schleife. Dadurch wird immer Code in Form von Daten an die jeweiligen Operationen übergeben und, damit zusammenhängend, was mit dem Datenstrom geschehen soll. Wie das Ganze ausgeführt werden soll, bleibt dabei den Stream-Implementierungen vorbehalten. Man kann sich dadurch auf das Wesentliche beschränken, nämlich lesbaren, leicht wartbaren und wiederverwendbaren Code zu schreiben.[Goe13b]

4.4 Erzeugung von Streams

Es werden von Java mehrere Möglichkeiten zur Verfügung gestellt um einen Stream zu erzeugen. Jeder Stream benötigt dabei eine Art von Datenquelle. Im Interface *Collections* werden zwei neue *default*-Methoden festgelegt, um aus einer Collection einen Stream zu erstellen. So kann mit Hilfe der Methode *stream* ein sequenzieller, und mit *parallelStream* ein paralleler Stream zurückgegeben werden. Eine wichtige Rolle spielt hier die Klasse *StreamSupport*, welche Hilfsmethoden zur Erzeugung und Manipulation von Streams bereitstellt. Diese verwendet man jedoch eher bei der Programmierung von Bibliotheken.[Orab]

Weitere Möglichkeiten bieten die statischen Methoden des *Stream*-Interfaces. So kann zum Beispiel mit *Stream.of* ein Stream von einem oder mehreren Objekten eines generischen Typs T erstellt werden. Mit Hilfe von *Stream.iterate* lassen sich auch unendliche Folgen von geordneten Objekten erzeugen.

```
//Signatur der statischen Methode iterate
public static<T> Stream<T> iterate(final T seed,
                                   final UnaryOperator<T> f)

//Beispiel für einen unendlichen Stream
Stream.iterate(0, x->x+2)
      .limit(10)
      .forEach(System.out::println);
```

Um einen unendlichen Stream zu erzeugen, nimmt die statische Methode *iterate* als erstes Argument einen Identitätswert vom Typ T entgegen. Über den *UnaryOperator<T>*, der einen Spezialfall einer Funktion darstellt, welche einen Eingangsparameter vom Typ T entgegen nimmt und dessen Resultat ebenfalls vom Typ T ist, wird zuerst ausgehend vom Identitätswert und anschließend vom vorherigen Wert der nachfolgende Wert berechnet. Im Beispiel oben wird über *Stream.iterate* der unendliche Datenstrom der geraden Zahlen repräsentiert. Dieser Stream terminiert jedoch nach endlicher Zeit nach Ausgabe der ersten zehn geraden Zahlen auf Grund des terminierenden Operators *limit*. Sofern man einen korrekten UnaryOperator angibt, sich also an die Vorgabe hält, dass der Eingangstyp gleich dem Ausgangstyp entspricht(*Function<T,T>*), so können auch unendliche Streams

von komplexeren Objekten erzeugt werden, beispielsweise ein unendlicher Stream von Objekten der Klasse *Person*.

Mit Hilfe von *Stream.generate*, der als Argument einen *Supplier* entgegennimmt, können ungeordnete, unendliche Streams von konstanten Zahlen oder beispielsweise von Zufallszahlen erstellt werden. Über *Stream.concat* lassen sich Streams zusammenfügen. Des Weiteren bietet das Interface *Stream* noch ein *Builder*-Interface, über welches man manuell einen Stream von Objekten erstellen kann. Dadurch kann man sich den Overhead einsparen, der beispielsweise bei der Erzeugung eines Arrays entsteht. Bei den Streams für primitive Typen können über die statischen Methoden von *IntStream* und *LongStream* mit der Methode *range(start,end)* ein Stream vom Startwert bis zum Endwert erzeugt werden.[Orab]

Es gibt jedoch auch außerhalb der *Collections API* und der neuen *Streams API* Möglichkeiten Streams zu erzeugen. So können Arrays über die neu hinzugekommene Methode *Arrays.stream* in einen Stream umgewandelt werden. Auch im Bereich von Dateien lassen sich Streams erzeugen. So kann einerseits beim Lesen einer Datei durch einen *BufferedReader* mit *BufferedReader.lines* ein Stream vom Typ *String* angefordert werden, und andererseits kann beispielsweise von einem Dateipfad über die Methode *list* ein *Stream* vom Typ *Path* erzeugt werden. Man sieht also, dass sich die Entwickler von Java 8 hier besonders viel Mühe gegeben haben, die Erzeugung von Streams über verschiedene Methoden zu ermöglichen und dadurch die Verwendung dieser deutlich einfacher zu gestalten.[Orab]

4.5 Stream-Operationen

In diesem Unterkapitel soll ein Überblick über die verschiedenen Arten von Operatoren theoretisch und anhand von Beispielen gegeben werden. Wie bereits erwähnt besitzen Streams immer eine Datenquelle. Diese kann über die oben erwähnten Methoden in einen Stream umgewandelt werden. Nun können Operationen auf jene Objekte, welche sich im Stream befinden, angewandt werden. Die Datenquelle zusammen mit den Operationen wird als *Stream Pipeline* bezeichnet. Innerhalb dieser folgen auf die Datenquelle beliebig viele Zwischenoperationen, gefolgt von einer terminierenden Operation.

Es wird grundlegend zwischen Zwischenoperationen und terminierenden Operationen auf Streams unterschieden. Die Zwischenoperationen sind anhand ihrer Signatur zu erkennen. Das Resultat einer solchen Operation ist nämlich immer ein Stream. Dieser kann vom selben Typ sein, beispielsweise bei der Anwendung der Filter-Operation, kann jedoch auch auf einen anderen Typ umgewandelt worden sein, beispielsweise über die Operation *map*. Bei Zwischenoperationen erfolgt die Auswertung immer verzögert. Die Operationen werden also erst dann ausgewertet, wenn die Ergebnisse der Auswertung benötigt werden. Benötigt werden sie erst dann, wenn die am Ende stehende terminierende Operation durchgeführt wird.

Terminierende Operationen werden bei Ausführung der Methode im Gegensatz zu Zwi-

schenoperationen strikt ausgewertet. Dabei werden die Daten zuerst durchlaufen und die jeweilige Operation ausgeführt, bevor ein Ergebnis geliefert wird. Als Ausnahmen gelten hier die Methoden *iterator* und *spliterator*. Hier wird das Durchlaufen der Pipeline an den Client übergeben, für den Fall dass die bereitgestellten Operationen für die Verarbeitung der Objekte des Streams nicht ausreichen. Terminierende Operationen sind ebenfalls anhand ihrer Signatur zu erkennen. Entweder liefern sie kein Resultat zurück, wie beispielsweise bei *forEach*. Dann werden jedoch Seiteneffekte wie zum Beispiel I/O-Operationen durchgeführt. Oder sie liefern als Resultat etwas anderes als einen Stream zurück. So können zum Beispiel bei *collect* Streams wieder in eine beliebige Collection umgewandelt werden, oder mittels *reduce* die Summe eines *IntStreams* berechnet werden.[Orab]

4.5.1 Verzögerte und strikte Auswertung von Operationen

Im folgenden Beispiel soll die Auswertung von Zwischenoperationen und terminierenden Operationen veranschaulicht werden.

```
IntStream stream = IntStream.rangeClosed(0,5);

stream.filter( x -> {
    System.out.print(" F"+x);
    return x < 3;
}).forEach(x -> System.out.print(" A"+x);
```

In diesem Beispiel wird ein *IntStream* von null bis fünf erzeugt. Die hier verwendete Methode *rangeClosed* verhält sich wie *range*, nur dass der Endwert noch inklusiv angegeben ist, fünf ist also auch noch im Stream vertreten. Man sollte möglichst auf Seiteneffekte innerhalb der Lambda-Ausdrücke, welche den Stream-Operationen übergeben werden, verzichten. In diesem Beispiel soll dies nur zu Debugzwecken eingesetzt werden, um die verzögerte Auswertung der Zwischenoperationen besser erklären zu können. Statt *println* direkt in den Funktionskörper des Lambda-Ausdrucks zu schreiben, wäre es auch möglich, die Stream-Operation *peek* zu verwenden. Diese nimmt ebenfalls einen *Consumer* entgegen. Des Weiteren wird in [Orab] darauf hingewiesen, dass *println* für das Debuggen normalerweise kein Problem darstellen sollte. In diesem Beispiel sollen nur jene Zahlen, welche kleiner als Drei sind ausgegeben werden. Das *F* gefolgt von der Zahl steht dabei dafür, dass die Zahl innerhalb der Methode *filter* überprüft wird, und das *A* bei *print* repräsentiert dabei die Ausgabe innerhalb der *for-each*-Schleife.

Würde die Auswertung der Zwischenoperation *filter* nicht verzögert, bei Bedarf, sondern strikt erfolgen, würde man von der folgenden Ausgabe ausgehen können.

```
//Bei strikter Auswertung von Zwischenoperationen
//Wenn es die verzögerte Auswertung nicht geben würde
F0 F1 F2 F3 F4 F5 A0 A1 A2
```

Zuerst würden die Werte von null bis fünf gefiltert werden, daher die Ausgabe F0 bis F5 und anschließend würden die Werte, die kleiner als Drei sind, ausgegeben werden, also A0 bis A2. Hier sei nochmals erwähnt, dass diese Ausgabe nicht der Realität entspricht. Lässt man dieses Beispiel in Java 8 durchlaufen, so bekommt man folgende Ausgabe zu sehen.

```
//Tatsächliche Ausgabe wegen verzögerter Auswertung
F0 A0 F1 A1 F2 A2 F3 F4 F5
```

Hier verdeutlicht sich die verzögerte Auswertung innerhalb der Zwischenoperationen, hier im Beispiel der Filter-Operation, angewandt auf Streams. Erst nachdem mit der Methode *forEach* eine terminierende Operation ausgeführt wird, werden die Werte der Filter-Operation benötigt. Das bedeutet, die Filter-Operation wird erst jetzt durchgeführt. Und da der Stream, zusammen mit seinen Operationen eine Pipeline darstellt, wird das erste Objekt des Streams auf die Eigenschaft *kleiner als drei* überprüft und wenn diese erfüllt ist, so wird der Wert an die terminierende *forEach*-Operation weitergeleitet. Anschließend wird das Objekt vom Typ *int* über den Lambda-Ausdruck an die Methode *println* weitergereicht. Es handelt sich hierbei um eine terminierende Operation, weil der Rückgabewert keinen Stream darstellt. Die Operation *forEach* nimmt lediglich einen *Consumer* entgegen und liefert kein Resultat zurück.

4.5.2 Zwischenoperationen in Streams

Zwischenoperationen können in zustandsbehaftete und zustandslose Zwischenoperationen unterteilt werden. Die zustandslosen Operationen benötigen keine Informationen von vorherig ausgewerteten Objekten innerhalb des Streams. Sie haben daher den Vorteil, unabhängig voneinander ausgewertet werden zu können. Diese Eigenschaft hat gerade bei der parallelen Auswertung einer Stream-Operation den Vorteil, dass man sich keine Gedanken bei der Aufteilung auf die einzelnen Threads machen muss. Des Weiteren spielt auch die Reihenfolge der Auswertung auf Grund der Unabhängigkeit der Elemente keine Rolle. Außerdem können, unabhängig davon wie viele zustandslose Zwischenoperationen man aneinander hängt, alle in einem Durchlauf des Streams verarbeitet werden können. Hingegen muss man bei zustandsbehafteten Zwischenoperationen, wie zum Beispiel *sorted*, den gesamten Stream durchlaufen um ein Ergebnis zu bekommen. Sie heißen deshalb zustandsbehaftet, weil für die Auswertung einer Operation auf einem Element des Streams die Auswertung eines anderen Elementes des Streams notwendig sein kann. So muss man beispielsweise für *sorted* alle Objekte innerhalb des Streams betrachten, um sie auch korrekt sortieren zu können. Ebenso verhält es sich bei *distinct*. Dadurch kann es bei der parallelen Auswertung zustandsbehafteter Zwischenoperationen passieren, dass

zusätzlicher Speicher benötigt wird, um relevante Werte zwischenzuspeichern. Ebenso kann es zu mehrmaligem Durchlauf der Elemente des Streams kommen, woduch eine sequenzielle Verarbeitung performanter sein könnte.[Orab]

Filter

Hier werden die verschiedenen Arten von Filtern vorgestellt. Bei der Operation *filter* handelt es sich um eine zustandslose Zwischenoperation.

```
Stream<T> filter(Predicate<? super T> predicate);
```

Die Methode *filter* nimmt als Argument das funktionale Interface *Predicate* entgegen. Diese erlaubt uns einen Stream von Objekten auf gewisse Eigenschaften zu überprüfen. Der Eingangsparameter des Lambda-Ausdrucks muss dabei einen Obertypen von T darstellen, wobei T der Typ-Parameter des Streams ist. Da es sich um eine Zwischenoperation handelt, wird ein Stream zurückgegeben, auf den *filter* angewandt wurde. In diesem Fall hat der resultierende Stream denselben Typ wie der Eingangs-Stream und besteht meist aus weniger Elementen.

Die Operation *distinct* liefert einen Stream zurück, der keine doppelten Werte enthält. Es handelt sich daher um eine Zwischenoperation. Man könnte *distinct* daher als eine Art Spezialfall von *filter* ansehen, da die doppelten Werte herausgefiltert werden. Für geordnete Streams wird das erste Element behalten und alle anschließenden Duplikate werden verworfen. Diese Eigenschaft wird als Stabilität bezeichnet. Wenn man diese Operation innerhalb eines parallelen Streams verwendet, sollte man darauf achten, ob die Elemente des Streams geordnet oder ungeordnet sind. Stabilität im Zusammenhang mit parallelen Streams ist relativ teuer, da Zwischenspeicherung notwendig wird. Müssen die Elemente also nicht unbedingt geordnet sein, so empfiehlt es sich, aus Gründen der Performance den Stream mittels *unordered* explizit als ungeordneten Stream zu deklarieren. Dies werden wir im Performance Kapitel näher untersuchen.[Orab]

Map und Flatmap

Um innerhalb eines Streams dessen Objekte zu manipulieren, können *map* und *flatmap* eingesetzt werden. Zuerst werden die Funktionsweisen beider Operationen vorgestellt. Anschließend werden die Signaturen betrachtet und anhand von Beispielen sollen die Operationen verständlich gemacht werden. Beide Operationen nehmen einen Lambda-Ausdruck vom Typ des funktionalen Interfaces *Function* entgegen. Je nach angegebener Funktion kann sich der Typ der Objekte im resultierenden Stream vom Typ der Objekte des Eingabe-Streams unterscheiden. Zusätzlich gibt es für die primitiven Datentypen *int*, *double* und *long* spezielle Methoden wie beispielsweise *mapToInt*, wobei der resultierende Stream ein *IntStream* ist. Hier sehen wir die Signaturen der beiden Operationen.

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);

<R> Stream<R> flatMap(
    Function<? super T, ? extends Stream<? extends R>> mapper);
```

Bei der Operation *map* wird über die angegebene Funktion jedes Element des Streams vom Typ T in ein Element vom Typ R, beziehungsweise in ein Element des Untertyps von R umgewandelt. Die Operation *flatMap* funktioniert nach einem sehr ähnlichen Prinzip. Es wird ebenfalls eine Funktion als Argument angenommen. Ebenso wird ein Stream von Objekten des Typs R zurückgeliefert. Es gibt jedoch einen entscheidenden Unterschied. Und zwar wird über die Funktion, die an *flatMap* übergeben wird, jedes Element in einen Stream von Elementen umgewandelt. So wird ein Element vom Typ T in beliebig viele Elemente des Typs R umgewandelt, wobei die Elemente als Datenquelle für einen Stream vom Typ R dienen. Abschließend werden die Elemente der dadurch entstandenen Streams in einem einzigen Stream gebündelt und von *flatMap* zurückgegeben. Das folgende Beispiel soll die Funktionsweise von *map* und *flatMap* verdeutlichen. Dadurch soll klar werden, welcher der beiden Operationen in welcher Situation vorteilhafter einzusetzen ist.

Im nächsten Beispiel gehen wir von folgendem Szenario aus. Es geht um eine Gruppe von Personen, die als Liste angelegt wird. Jede Person besteht dabei aus einem Namen, dem Alter und beispielsweise aus einer Menge von Hobbys. Diese werden durch eine Liste bestehend aus Strings repräsentiert. Im ersten Beispiel sollen alle Namen von Personen ausgegeben werden, welche älter als achtzehn Jahre sind.

```
persons.stream()
       .filter(p -> p.getAge() >= 18)
       .map(Person::getName)
       .forEach(System.out::println);
```

Die Liste der Personen *persons* wird zuerst über die Methode *stream* als Datenquelle für unseren Stream angegeben. Anschließend werden die Personen über das Prädikat *p -> p.getAge() >= 18* gefiltert. Da nur die Namen und nicht die Objekte ausgegeben werden sollen, erfolgt über *map* eine Umwandlung von *Person* auf *String*, wobei die Personen nun innerhalb des resultierenden Streams über ihren Namen repräsentiert werden. Anschließend wird die Berechnung des Streams über *forEach* gestartet.

Im nächsten Beispiel sollen jene Hobbys aller Personen ausgegeben werden, deren Hobbys mit R beginnen. Zuerst wird dies mit *map* versucht. Es gibt mehrere Möglichkeiten dieses Problem nur mittels *map* zu lösen. Jedoch sehen alle Lösungsvorschläge nicht wirklich sauber implementiert aus. Die Lesbarkeit und Wartbarkeit des Codes wird dabei vernachlässigt. Das Problem liegt darin, dass *Person.getHobbys* keine einzelnen Elemente zurückliefert, sondern eine Liste von Strings, welche die Hobbys der jeweiligen Person repräsentieren. Eine Lösung mit *map* wäre beispielsweise die doppelte Anwendung der *forEach*-Operation und verschachtelten Streams.

```
persons
  .stream()
  .map(p -> p.getHobbys()
             .stream())
             .forEach(s -> s.filter(h -> h.startsWith("R"))
                            .forEach(System.out::println));
```

Dabei wird die Liste, die von *getHobbys* zurückgeliefert wird, mit der Methode *stream* in einen Stream umgewandelt und anschließend wird für jeden Stream der dadurch entstanden ist nach dem angegebenen Prädikat gefiltert und danach für jedes Element *println* aufgerufen. Das Problem bei dieser Variante ist, dass es sogar bei diesem vergleichsweise einfachen Beispiel schon schwieriger wird diesen Code zu lesen. Daher wurde eben für solche Situationen, wo mehrschichtig hierarchische Objekte zurückgegeben werden, die Methode *flatMap* hinzugefügt. Dadurch vereinfacht sich das Beispiel folgendermaßen.

```
persons.stream()
       .filter(p -> p.getAge() >= 18)
       .flatMap(p -> p.getHobbys().stream())
       .filter(h -> h.startsWith("R"))
       .forEach(System.out::println);
```

Auch hier wird über die Liste zuerst mittels der Methode *stream* ein Stream von Personen erzeugt. Da *flatMap* jedoch nach der Umwandlung von Person in Streams vom Typ *String* alle erzeugten Streams zusammenfügt, erspart man sich das verschachtelte *forEach*. Dadurch werden alle Operationen nur auf einen Stream angewandt und die ursprüngliche Lesbarkeit der aneinander geketteten Stream-Operationen ist wieder vorhanden. So liest sich die Lösung des Problems durch die Aneinanderkettung der Operationen wie die Problemstellung. *Filtere zuerst die Personen die älter als 18 sind, gib mir anschließend die Hobbys der Personen zurück und filtere diese nach dem Anfangsbuchstaben R und gib diese abschließend über println in der Konsole aus.*

4.5.3 Terminierende Operationen

In diesem Unterkapitel wollen wir die terminierenden Operationen von Streams näher vorstellen. Dazu gehören die Operationen *reduce* und *collect* und die Gruppe der Kurzschlussoperationen, über die es ermöglicht wird, die Verarbeitung von endlichen und unendlichen Streams vorzeitig abzubrechen.

Reduce

Mit Hilfe von *reduce* können die Elemente eines Streams auf einen Wert reduziert werden. So kann beispielsweise auf einem *IntStream* durch Reduktion die Summe der Elemente vom Typ *int* berechnet werden. Dazu wird ein Startwert benötigt, zusätzlich dazu

wird eine Akkumulator-Funktion übergeben. Bei der dritten Variante, wobei *reduce* auf einen anderen Typen als jenen der Stream-Objekte umwandelt, wird zusätzlich eine Funktion zum Kombinieren der Werte für die parallele Verarbeitung der Daten benötigt. Nachfolgend sieht man die drei Signaturen der überladenen Methode *reduce*. Im Anschluss daran folgt eine kurze Erklärung der einzelnen Methoden und ein Beispiel soll die Funktionsweise verdeutlichen.[Orab]

```
T reduce(T identity, BinaryOperator<T> accumulator);

Optional<T> reduce(BinaryOperator<T> accumulator);

<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner);
```

Die erste Variante nimmt als erstes Argument (*identity* vom Typ T) den Identitätswert der Reduktion entgegen. Als zweites Argument wird ein Akkumulator in Form eines Lambda-Ausdrucks übergeben, der das funktionale Interface *BinaryOperator<T>* repräsentiert. Über diesen Lambda-Ausdruck wird bestimmt, wie die Elemente des Streams reduziert, beziehungsweise zusammengefaltet werden. Bei der Akkumulator-Funktion ist es wichtig, dass es sich dabei um eine assoziative, nicht-einmischende und zustandslose Funktion handelt. Ohne diese Eigenschaften ist eine parallele Abarbeitung des *reduce*-Operators nicht möglich, da die Ergebnisse bei einer parallelen Verarbeitung der Daten verfälscht werden können. Bei der Eigenschaft der Assoziativität geht es darum, dass bei der parallelen Verarbeitung die Teilergebnisse zu einem Ganzen kombiniert werden müssen. Nehmen wir als Beispiel dazu die Addition her, so haben wir auf Grund der Assoziativität kein Problem beim Zusammenfügen der Teilergebnisse, da $a + b$ dasselbe Ergebnis zurückliefert wie $b + a$. Wenn wir hier als Operation für den Akkumulator die Subtraktion verwenden würden, würden wir je nach Durchlauf unterschiedliche, größtenteils falsche Ergebnisse erhalten da $a - b$ nicht dasselbe Ergebnis zurückliefert wie $b - a$. Durch die nicht-einmischenden und zustandslosen Funktionen können diese unabhängig voneinander abgearbeitet werden. Dadurch kann die parallele Ausführung vereinfacht werden, da man nicht auf Synchronisation wegen gleichzeitigem Schreibzugriff auf Variablen achten muss.

Das nächste Beispiel zeigt, wie man die Summe eines *IntStreams* mit Hilfe von *reduce* berechnen kann. Dabei gehen wir von einem beliebigen, jedoch endlich großen IntStream aus.

```
intStream.stream()
         .reduce(0, (x,y) -> x+y);
```

Als Identitätswert wird die Zahl null angegeben. Von diesem Wert ausgehend werden die Elemente des Streams nach und nach mit dem Funktionskörper des Lambda-Ausdrucks

x+y auf deren Summe reduziert. Es gibt für häufiger verwendete Reduktionen auch Spezialfälle, welche direkt über Methodenreferenz angegeben werden können. So gibt es unter anderem für die Summe, den minimalen und maximalen Wert und den Durchschnittswert entsprechende Methoden in Java 8. Je nachdem auf welchen Typen diese angewandt werden, muss eventuell noch ein *Comparator* definiert werden, um festzulegen wie die Elemente miteinander verglichen werden können. In diesem Beispiel könnte man also *(x,y) -> x+y* durch *Integer::sum* ersetzen. Da wir hier auf einem *IntStream* operieren, könnte man auch die Operation *sum*, als Spezialfall von *reduce* verwenden.[Orab]

Als zweite Variante der überladenen Methode gibt es auch ein *reduce* ohne den Identitätswert. Dabei wird das erste Element als Startwert hergenommen. Da der Stream aber leer sein könnte, wird der Wert in ein *Optional*-Objekt gekapselt. Die genaue Funktionsweise des neu hinzugekommenen *Optional* wird im Zuge des Vergleiches zwischen Haskell und Java in einem späteren Kapitel dieser Arbeit genauer erklärt.

Bei der dritten Variante wird, im Gegensatz zu den anderen beiden, nicht auf denselben Typ der Objekte des Streams reduziert, sondern auf einen anderen beliebigen Typ. Daher hat der Startwert für den Akkumulator den Typ U, wenn wir von einem Stream von Objekten des Typs T ausgehen. Der Akkumulator ist hier kein *BinaryOperator* sondern eine *BiFunction* von (U,T) nach U. Dieser nimmt den aktuellen Wert der Reduktion vom Typ U als erstes Argument und das nächste Element des Streams vom Typ T als zweites Argument entgegen und liefert die Reduktion dieser als Typ U zurück. Für die parallele Auswertung dieser *reduce*-Variante wird noch *BinaryOperator* vom generischen Typ T als Kombinierer angegeben. Dieser muss wie der Akkumulator assoziativ, non-interfering und zustandslos sein, um eine korrekte parallele Auswertung zu garantieren. Er wird dazu verwendet um die Ergebnisse einzelner Threads bei der parallelen Auswertung zusammenzufügen. Bei der sequenziellen Auswertung wird er nicht verwendet.

Im folgenden Beispiel sollen die ECTS aus einer Liste von Lehrveranstaltungen zusammengerechnet werden. Dazu wird eine Klasse Lehrveranstaltungen verwendet, welche neben anderen Informationen die Anzahl der ECTS-Punkte als *int* angibt.

```
lvas.stream()
    .reduce(0,
            (Integer x, Lehrveranstaltung l) -> x + l.getEcts(),
            (Integer x, Integer y) -> x + y));
```

Der Stream ist vom Typ *Lehrveranstaltung* und die ECTS-Punkte sind als *int* innerhalb der Klasse angegeben. Als Startwert für die Reduzierung wird auch hier die Zahl null angegeben. Die Typen für die Parameter der Lambda-Ausdrücke sind für die bessere Verständlichkeit des Beispiels explizit angegeben. Der Compiler könnte die Typen hier ohne weiteres ableiten. Als zweites Argument für *reduce* wird ein Lambda-Ausdruck vom Typ *BiFunction<U,T,U>* erwartet. U ist hier *Integer* und T entspricht *Lehrveranstaltung*. Der Akkumulator legt also fest, dass zum vorherigen Wert die ECTS des aktuellen

Stream-Objekts Lehrveranstaltung hinzugezählt werden. Daher ist der Rückgabewert der *BiFunction* ebenfalls vom Typ *Integer*. Abschließend wird für die parallele Ausführung der Kombinierer festgelegt, wobei es sich dabei um einen *BinaryOperator* vom Typ *U* handelt. Dabei wird festgelegt wie die einzelnen Ergebnisse der verschiedenen Threads bei der parallelen Abarbeitung des Streams wieder zusammengefügt werden. Hier werden einfach die jeweiligen Summen der Teil-Ergebnisse addiert. Vereinfacht könnte man in diesem Beispiel als Kombinierer die Methoden-Referenz *Integer::sum* angeben. Man könnte das vorherige Beispiel auch folgendermaßen darstellen.

```
lvas.stream()
    .map(l -> l.getEcts())
    .reduce(Integer::sum));
```

Wie man sieht, ist die dritte Variante durch eine Umwandlung der Elemente des Streams auf einen anderen Typ und eine anschließende Reduktion dieser als Alternative durchzuführen. Die Vorteile *map* und *reduce* zu verwenden, ist die höhere Lesbarkeit des Codes. Es gibt jedoch Szenarien, bei denen die *reduce*-Variante mit drei Argumenten, bei der das Mapping inklusive durchgeführt wird, eine höhere Performance liefert. Dies ist der Fall wenn man durch die vorhergehende Reduktion eines Elements beim aktuellen Element im Stream Berechnungsaufwand einsparen kann.[Orab]

Kollektoren

Bei der Operation *collect* handelt es sich um eine terminierende Operation. Mit dessen Hilfe kann der Stream in eine beliebige *Collection* umgewandelt werden. Es handelt sich bei dieser Operation ebenfalls um eine Reduktion, jedoch wird hier von einer veränderbaren Reduktion gesprochen, weil die Elemente des Streams in eine Collection gepackt werden. Dabei wird nicht bei jedem Element ein neues Resultat erzeugt, sondern die bestehender Datensatz wird um das jeweilige Element erweitert. Daher handelt es sich auch um eine Reduktion mit Veränderung der Daten. Es befinden sich zwei überladene Methoden *collect* im Interface *Stream*. Diese werden nun vorgestellt.[Orab]

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

Bei dieser Variante werden Versorger, Akkumulator und Kombinierer einzeln, in Form von Lambda-Ausdrücken, der Methode übergeben. Der Versorger liefert den neuen Container des jeweiligen Datensatzes zurück. Der Akkumulator legt wie bei *reduce* fest, wie die Elemente des Streams reduziert werden sollen. Der Kombinierer wird für die parallele Verarbeitung der Operation benötigt, um die parallel gelösten Teilaufgaben zusammenzufügen. Im Vergleich zu *reduce* liefert der Akkumulator und der Kombinierer jedoch kein Resultat zurück, daher verwenden beide einen *BiConsumer*.

```
persons.stream()
       .filter(p -> p.getAge() >= 18)
       .collect(() -> new ArrayList<>(),
                (l, p) -> l.add(p),
                (list1, list2) -> list1.addAll(list2)));
```

In diesem Beispiel sollen alle Personen, die über 18 Jahre alt sind, gefiltert und anschließend in einer *ArrayList* gesammelt werden. Als Versorger wird hier ein Lambda-Ausdruck verwendet, der eine neue ArrayListe mit Personen zurückliefert. Der Typ der Elemente der ArrayList wird über den Compiler inferiert, daher muss dieser nicht explizit zwischen <> angegeben werden. Der Akkumulator beschreibt wie die Elemente des Streams zur ArrayList hinzugefügt werden sollen. Dabei wird die eine Person *p* über *l.add(p)* zur Array-Liste *l* hinzugefügt. Als Kombinierer wird die Methode *addAll* von der Klasse *ArrayList* verwendet. Da bei allen drei Lambda-Ausdrücken lediglich die Argumente an eine bereits bestehende Methode weitergereicht werden, kann dieses Beispiel, mit Hilfe von Methoden-Referenzen vereinfacht werden.

```
persons.stream()
       .filter(p -> p.getAge() >= 18
       .collect(ArrayList::new,
                ArrayList::add,
                ArrayList::addAll);
```

Zusätzlich zu dieser Variante, wo Versorger, Akkumulator und Kombinierer einzeln angegeben werden, gibt es als Alternative das in Java 8 hinzugefügtes Interface *Collector*. Dementsprechend kann man der Operation *collect* auch eine Implementierung des Collector-Interfaces übergeben.

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

Auffallend sind hier die beiden generischen Typen R und A. Das Collector Interface besteht aus vier Funktionen. Wie bei der ersten Variante sind die ersten drei Funktionen der Versorger, Akkumulator und Kombinierer. Über die vierte Funktion, welche als *finisher* bezeichnet wird, kann zusätzlich, nach der Reduktion der Objekte des Streams, eine Umwandlung der Elemente der Collection stattfinden. Diese ist jedoch optional. Einige Implementierungen des Interfaces *Collector* werden von der Klasse *Collectors* bereitgestellt, wobei hier häufiger verwendete Kollektoren implementiert wurden. Neben Methoden wie *Collectors.toList()*, welche die Elemente des Streams in eine Liste umwandelt, gibt es mit *groupingBy* und *partitioningBy* zwei sehr interessante Implementierungen, um die Elemente eines Streams auf beispielsweise eine Map mit Schlüsseln und Werten aufzuteilen, beziehungsweise zu gruppieren. Bei beiden Methoden handelt es sich um überladene Methoden. Da es jeweils eine Variante gibt, die als letztes Argument einen Kollektor

entgegennimmt, können diese geschachtelt werden. Dadurch kann beispielsweise nach mehreren Kriterien gruppiert werden, oder aber innerhalb der Gruppierung Reduktionen durchgeführt werden. Diese beiden Methoden werden nun genauer betrachtet.

Über die statische Methode *groupingBy* in der Klasse *Collectors* können Kollektoren zurückgegeben werden, über die es möglich gemacht wird, die Elemente eines Streams in eine Map zu gruppieren. Dabei wird über das funktionale Interface *Function* eine Funktion zur Klassifikation des Schlüssels der Map festgelegt. Über diesen Schlüssel werden die Elemente des Streams dann gruppiert. So könnte man beispielsweise die Personen nach Anfangsbuchstaben gruppieren.[Sub14]

```
Map<Character,List<Person>> map =
    persons.stream()
           .collect(groupingBy(p -> p.getName().charAt(0)));

//Beispielausgabe
{A=[Anton 17, Alice 19], B=[Benni 18, Bob 21],
 C=[Chris 18, Conny 16]}
```

Der Lambda-Ausdruck *p -> p.getName().charAt(0)* legt die Funktion fest, über welche die unterschiedlichen Schlüssel der Map erzeugt werden. Die Schlüssel für diese Map sind daher vom Typ *Character*, da nur der erste Buchstabe des Strings betrachtet wird. Als Werte werden die Personen in einer Liste angegeben. Die Beispielausgabe zeigt die Personen durch ihren Namen und ihr Alter repräsentiert.

Mit Hilfe der statischen Methode *partitioningBy* wird ein Kollektor zurückgegeben, über den die Elemente des Streams in zwei Partitionen unterteilt werden. Dabei übergibt man der Methode einen Lambda-Ausdruck vom Typ *Predicate*, und die Elemente des Streams werden dann, sofern sie die Eigenschaft erfüllen, an die Liste mit dem Schlüssel *true* angehängt, und wenn das Element die Eigenschaft nicht erfüllt, an den Schlüssel *false* angehängt. Es wird also standardmäßig eine *Map<Boolean,List<T>* zurückgegeben. So können im folgenden Beispiel die Personen in zwei Listen aufgeteilt werden. Die eine Liste beinhaltet alle Personen unter 18 Jahren, und die zweite Liste beinhaltet alle Personen über 18.

```
Map<Boolean,List<Person>> map = persons.stream()
       .collect(partitioningBy(p -> p.getAge() < 18));
```

Da beide Methoden *groupingBy* und *partitioningBy* als letztes Argument einen Kollektor entgegennehmen können, gibt es auch die Möglichkeit, Multilevel-Maps zu erzeugen. Dabei werden mehrere Maps ineinander geschachtelt. Dadurch kann die Gruppierung über mehrere Schlüssel spezieller festgelegt werden. Zum Beispiel könnte man die Personen aus dem vorherigen Beispiel zuerst in männliche und weibliche Personen gruppieren und

anschließend noch über den Anfangsbuchstaben als zweiten Schlüssel Untergruppierungen schaffen. Zusätzlich soll nur der Name der Person als Wert in der Map eingetragen werden.[Sub14]

```
Map<String, Map<Character,List<String>>> map =
    persons.stream()
           .collect(groupingBy(Person::getGender,
                    groupingBy(p -> p.getName().charAt(0),
                    mapping(Person::getName,toList()))));
```

In diesem Beispiel soll eine Multilevel-Map erzeugt werden, wobei als erster Schlüssel das Geschlecht der Person und als zweiter Schlüssel der Anfangsbuchstabe des Namens verwendet wird. Als Wert soll der Name der Person eingetragen werden. Zuerst wird aus der Liste der Personen ein Stream erzeugt. Dieser wird über *collect* in eine Map umgewandelt. Um die erste Ebene der Map zu erzeugen, werden die Personen über die Methoden-Referenz *Person::getGender* auf männlich und weiblich aufgeteilt, hier abgekürzt durch den String *M* und *W*. Da *groupingBy* als zweites Argument einen weiteren Kollektor zulässt, können wir ein zweites *groupingBy* einsetzen, um eine weitere Ebene innerhalb der Map zu erzeugen. Als zweiten Schlüssel wird, über den Lambda-Ausdruck, der erste Buchstabe des Namens festgelegt. Es soll nur der Name der Person und nicht das gesamte Objekt *Person* in diesem Beispiel als Wert für die Multilevel-Map eingetragen werden. Daher wird noch der Kollektor *mapping* verwendet um über *Person::getName* nur den Namen abzubilden. Im Folgenden sieht man eine mögliche Ausgabe der Map durch *println*.

```
{W={A=[Alice],C=[Conny]},M={A=[Anton],B=[Benni,Bob],C=[Chris]}}
```

Kurzschluss-Operationen

Über Kurzschluss-Operationen kann die Abarbeitung von Operationen auf Streams frühzeitig fertiggestellt werden. Der Einsatz von Kurzschluss-Operationen bringt zwei Vorteile mit sich. Zum einen wird es dadurch ermöglicht, Operationen auf Streams mit endlicher Anzahl von Objekt-Referenzen schneller und effizienter durchzuführen. Zum anderen wird es dadurch erst möglich, auf unendlichen Streams zu operieren und ein Resultat zurückzubekommen. Man betrachte beispielsweise den folgenden unendlichen Stream, bestehend aus Elementen des primitiven Typs *int*. Nachdem der maximale Wert von *int* erreicht wird, wird auf Grund eines Overflows beim minimalen Wert von *int* weitergezählt, es handelt sich also um einen unendlichen Stream.[Orab]

```
IntStream stream = IntStream.iterate(0, i -> i+1);
stream.forEach(System.out::println);
```

Bei diesem Beispiel wird die Methode *forEach* angewandt. Jedes Element des Streams wird dabei an die Methode *println* weitergeleitet und von dieser konsumiert. Dieser unendliche Stream kann jedoch nicht terminieren, da *forEach* die Anzahl der Elemente nicht einschränkt und die Ausführung der Operation auf einen unendlichen Stream daher nicht in endlicher Zeit möglich ist. Um also mit unendlichen Streams arbeiten zu können, werden Kurzschluss-Operationen benötigt, ansonsten terminieren unendliche Streams nicht. Der Einsatz von Kurzschluss-Operationen sichert jedoch nicht zu, dass ein unendlicher Stream tatsächlich terminiert[Orab]. Dazu folgt am Ende dieses Unterkapitels ein Beispiel. Das folgende Beispiel zeigt den Einsatz eines Kurzschluss-Operators.

```
stream.limit(20)
      .forEach(System.out::println)
```

Die Methode *limit* mit der Signatur *Stream<T> limit(long maxSize)* gibt nur eine gewisse Anzahl an Elementen an den zurückgegebenen Stream weiter. Dadurch kann die Länge des Streams limitiert werden. In diesem Beispiel verkürzt *limit* den Stream mit unendlicher Länge auf eine Länge von 20 Elementen. Diese werden anschließend ausgegeben und die terminierende Operation *forEach* terminiert im Gegensatz zum vorherigen Beispiel dank der Kurzschluss-Operation tatsächlich. Im folgenden werden die anderen Kurzschluss-Operationen aufgelistet.

- *anyMatch* - Die Methode *anyMatch* nimmt ein Prädikat entgegen und überprüft, ob eines der Elemente innerhalb des Streams die angegebene Eigenschaft erfüllt. Sie liefert *true* zurück wenn die Eigenschaft von einem beliebigen Element erfüllt wird, ansonsten wird *false* zurückgeliefert. Sobald ein Element mit der gewünschten Eigenschaft gefunden wurde, kann das Durchlaufen des Streams unterbrochen werden und es wird *true* zurückgeliefert. Diese Operation entspricht dem Existenzquantor.
- *allMatch* - Verhält sich so wie *anyMatch*, jedoch wird hier überprüft, ob alle Elemente die übergebene Eigenschaft erfüllen. Diese Operation entspricht dem Allquantor (für alle Elemente gilt). Sobald ein Element die Eigenschaft nicht erfüllt, kann die Operation terminieren und *false* zurückgeben.
- *noneMatch* - Entspricht der Negierung von *allMatch*. Liefert *true*, wenn kein Element des Streams der Eigenschaft entspricht, ansonsten *false*. Sobald ein Element die Eigenschaft erfüllt, terminiert *noneMatch* und liefert *false* zurück.
- *findFirst* - Liefert das erste Element des Streams zurück. Als Rückgabewert wird *Optional<T>* zurückgegeben. Dies ist der neue Abstraktions-Mechanismus in Java 8 um Nullpointer-Exceptions zu vermeiden. Wird wie bereits erwähnt in einem späteren Kapitel genauer analysiert.
- *findAny* - Liefert ein beliebiges Element zurück. Diese Operation ist für parallele Streams gedacht, um schnellstmöglich ein Ergebnis zu liefern. Es handelt sich hierbei

um eine nicht-deterministische Operation. Das bedeutet, es können bei mehrmaliger Ausführung auf denselben Stream unterschiedliche Ergebnisse zurückgegeben werden.

Abschließend zu diesem Unterkapitel soll noch ein Beispiel gebracht werden, in dem die Kurzschluss-Operation nicht ausreichend ist, um Operationen angewandt auf Elemente eines unendlichen Stream terminieren zu lassen. Wir gehen wieder von dem unendlichen *IntStream* aus der weiter oben deklariert wurde.

```
stream.anyMatch( i -> false )
```

Dieses Beispiel wird trotz des Kurzschluss-Operators *anyMatch* nicht Terminieren, da *anyMatch* hier für jedes Element *false* zurückgibt. Hier wird also repräsentiert, dass kein Element des Streams die gewünschte Eigenschaft erfüllt. Wenn also ein Prädikat angegeben wird, dass für kein Element des unendlichen Streams erfüllt ist, so wird der unendliche Stream nicht terminieren. Man sollte sich daher beim Einsatz von Kurzschluss-Operatoren nicht auf die Terminierung der Streams verlassen.[Orab]

4.6 Fork-Join Framework

Bevor wir uns mit dem internen Aufbau von Streams und den dazugehörigen Spliteratoren beschäftigen, wollen wir uns das so genannte *Fork-Join-Framework* näher anschauen. Es kommt bei parallelen Streams im Hintergrund zum Einsatz. Zusätzlich dazu wollen wir einen generellen Blick auf einen Teil der Umsetzung paralleler Programmierung in Java werfen. Nachfolgend sehen wir die grundlegende Funktionsweise des Fork-Join Frameworks an Hand von Pseudo-Code. Es handelt sich dabei um eine parallele Variante des Teile und Herrsche Algorithmus, welcher im Englischen auch als *Divide and Conquer* bekannt ist.[Lea00]

```
Resultat löse(Problem problem) {
  if( problem ist klein genug ) {
    löse das Problem direkt
  } else {
    teile das Problem in unabhängige kleinere Teile auf
    erzeuge neue Teilaufgaben um die Probleme zu lösen (fork)
    warte auf Ergebnisse der Teilaufgaben
    füge die Ergebnisse der Teilaufgaben zusammen (join)
  }
}
```

Um ein gegebenes Problem zu lösen, wird es in kleinere Teilaufgaben aufgeteilt. Sind die Teilaufgaben klein genug, so können diese direkt gelöst werden. Andernfalls werden die

Teilaufgaben rekursiv in noch kleinere Teilaufgaben zerteilt. Abschließend müssen die Ergebnisse zusammengefügt werden, um so das gesamte Problem lösen zu können. Das Java Fork/Join-Framework wurde im Jahr 2000 vorgestellt und wurde schließlich im Jahr 2011 beim Release von Java 7 hinzugefügt. Um die erzeugten Teilaufgaben parallel lösen zu können müssen diese einzelnen Threads zugewiesen werden. Da die Erzeugung von neuen Threads aufwendig ist, werden meist so genannte Thread-Pools verwendet. Dabei werden die Threads nur einmal erzeugt und können dann wiederverwendet werden.

Parallele Programmierung in Java

Die älteste Form von paralleler Programmierung in Java basiert auf der Klasse *Thread* gemeinsam mit dem Interface *Runnable*. Die Funktionsweise wurde bereits am Anfang dieser Arbeit im Zusammenhang mit den funktionalen Interfaces besprochen. Über die Methode thread.start wird die Ausführung gestartet. Synchronisierung wird über thread.join erreicht, da hier der Main-Thread auf die Beendigung des neu erstellten Threads wartet, bevor die Ausführung fortgesetzt wird. Eine weitere Möglichkeit besteht darin, einen komplexeren Kontrollfluss über die Methoden notify() und wait() zu handhaben. Oder aber man verwendet synchronisierte Methoden über das Schlüsselwort *synchronized*. Doch dadurch könnte unter Umständen die Performance beeinträchtigt werden. Diese Varianten sind jedoch relativ hardwarenah und dadurch können sich leicht Fehler bei der Programmierung einschleichen.

Mit Java 5 kam das Paket *java.util.concurrent* um die parallele Programmierung in Java einfacher umsetzen zu können. So wurden unter anderem Executors und synchronisierte Collections zu Java hinzugefügt. Der Vorteil bei der Verwendung von Executors im Vergleich zu simplen Threads ist eine einfachere Verwaltung der gleichzeitig auszuführenden Tasks. Des Weiteren kann ein Executor Thread-Pools verwenden.

```
ExecutorService executorService =
    Executors.newFixedThreadPool(2);
executorService.invokeAll(Task1, Task2, Task3);
```

Hier können wir mittels *Executors.newFixedThreadPool* einen neuen ExecutorService erzeugen, wobei hier ein Thread-Pool mit maximaler Anzahl an verwendeten Threads als Parameter übergeben wird. Anschließend wird die Ausführung der Tasks an das ExecutorService übergeben. Diese werden von maximal zwei Threads abgearbeitet. Der dritte Task kann hier also nicht sofort ausgeführt werden, sondern wartet in einer Warteschlange auf dessen Ausführung. Die Tasks werden dabei asynchron ausgeführt. Ergebnisse erhält man über die Klasse *Future*. Diese repräsentiert das Resultat einer asynchronen Berechnung. Über die Methode *isDone* wird abgefragt ob die Berechnung bereits abgeschlossen wurde oder ob man noch auf das Ergebnis warten muss. Über *get()* kann das Ergebnis abgefragt werden. Diese Methode wartet falls notwendig auf das Ergebnis.

Durch die in Java 5 eingeführten Executors war es möglich, über die Verwendung des Callable Interfaces, so genannte *Divide and Conquer* beziehungsweise *map reduce* Algorithmen parallel in Java auszuführen und dessen Teilergebnisse zu einem Gesamtergebnis zusammenzufassen. Der Grund warum dennoch das Fork-Join-Framwork zu Java hinzugefügt wurde und warum Executors nicht ausreichend für Divide and Conquer Algorithmen sind ist Folgender. Die Aufteilung des Problems in kleinere Teile macht hier noch keine Probleme. Innerhalb der Callables können weitere Callables an den ExecutorService zur Ausführung hinzugefügt werden. Auf diese Teilergebnisse kann anschließend mit *get()* gewartet werden. Wenn jedoch der Aufwand einer Teilaufgabe höher ist als bei anderen Teilaufgaben dann müssen andere Threads auf das Ergebnis dieser Teilaufgabe warten. Dies könnte beispielsweise bei unausgeglichenen Bäumen oder Graphen eintreten. Dadurch geht quasi zwischenzeitlich Rechenleistung verloren beziehungsweise bleibt ungenutzt.[Pon11]

Dieses Problem wird durch das *Fork-Join-Framework* dahingehend gelöst, dass beim ForkJoin-ThreadPool unter den einzelnen Threads so genanntes Work-Stealing betrieben wird. Dabei werden zweiseitige Warteschlagen verwendet. Jeder Thread greift auf seine eigene Warteschlange über LIFO (last in first out) zu. Dabei wird wie bei einem Stack jeweils der jüngste Task (last in) zuerst aus der Warteschlange genommen (first out) und ausgeführt. Auf die Warteschlagen der anderen Threads wird über FIFO (first in first out) zugegriffen. Dabei wird der älteste Task von einem anderen Thread gestohlen und abgearbeitet. Wenn nun also ein Thread mit der Ausführung eines Tasks länger beschäftigt ist und die anderen Threads bereits mit der Ausführung ihrer Tasks fertig sind, so können diese auf die Warteschlange des einen Threads zugreifen und einen Task von ihm stehlen. Dadurch kann eine effizientere Ausführung erreicht werden.[Lea00][MVS09]

Fork-Join-Framework – Aufbau in Java

Eine wichtige Klasse des Fork-Join-Frameworks in Java ist der *ForkJoinPool*. Bei diesem handelt es sich um einen ExecutorService mit Work-Stealing. Diese Klasse stellt Methoden bereit um übergebene Tasks an den Thread-Pool zur Ausführung weiterzugeben. Die Aufgaben werden in Form von *ForkJoinTasks*, beziehungsweise deren Unterklassen *RecursiveAction* und *RecursiveTask* übergeben. Die einzelnen Aufgaben sollten blockende I/O Operationen vermeiden und nur auf unabhängige Variablen zugreifen. Dadurch kann die Synchronisation auf die Befehle *fork* und *join* reduziert werden. Wie bereits am Anfang dieses Kapitels erwähnt, wird mit *fork* eine neue Teilaufgabe an den ForkJoinPool übergeben. Mittels *join* wird auf die Fertigstellung der Unteraufgabe gewartet. *RecursiveAction* liefert im Gegensatz zu *RecursiveTask* kein Ergebnis zurück. Relevant für parallele Streams ist der in *ForkJoinPool* vorkommende *CommonForkJoinPool*. Dabei handelt es sich um den standardmäßig verwendeten ForkJoinPool des Fork-Join-Frameworks.[Orab]

```
// Abfrage der verwendeten Worker Threads für den Common Pool
// liefert bei einer 8-Kern-CPU 7 zurück
ForkJoinPool.getCommonPoolParallelism();
```

Der *CommonForkJoinPool* verwendet die Anzahl der CPU-Kerne minus eins. Es wird deshalb ein Thread weniger verwendet, weil der Aufrufer selbst auch schon einen Thread verwendet. Der aufrufende Thread wird hier ebenfalls für die Abarbeitung der Tasks verwendet. Dies lässt sich mit einem kleinen Experiment feststellen.

```
IntStream.iterate(0, x -> x+1)
         .parallel()
         .limit(10)
         .forEach( x -> System.out.println(
             "Thread: " + Thread.currentThread().getName()));

// Ausgabe:
Thread: ForkJoinPool.commonPool-worker-4
Thread: ForkJoinPool.commonPool-worker-3
Thread: ForkJoinPool.commonPool-worker-2
Thread: ForkJoinPool.commonPool-worker-6
Thread: ForkJoinPool.commonPool-worker-1
Thread: main
Thread: ForkJoinPool.commonPool-worker-2
Thread: ForkJoinPool.commonPool-worker-3
Thread: ForkJoinPool.commonPool-worker-4
Thread: ForkJoinPool.commonPool-worker-5
```

Über *Thread.currentThread()* können wir uns den ausführenden Thread ausgeben lassen. Hierbei lässt sich Folgendes feststellen. Auch der Main-Thread wird für die Ausführung des parallelen Streams mitverwendet. Manche Arbeiter-Threads des ForkJoinPools werden öfter für die Verarbeitung der Teilaufgabe verwendet. Bei einer Limitierung auf zehn Elemente werden beispielsweise nicht alle Threads verwendet.

```
Map<String, Long> map =
    Stream.iterate(0, x -> x+1)
          .parallel()
          .limit(100000)
          .collect(Collectors.groupingBy(
              x -> Thread.currentThread().getName(),
                  Collectors.counting()));

map.forEach((t,c) -> System.out.println("Thread:"+t+": "+c));

// Je nach Ausführung unterschiedliche Ergebnisse
// Ausgabe 1:
Thread: ForkJoinPool.commonPool-worker-1: 55968
Thread: ForkJoinPool.commonPool-worker-2: 3072
```

```
Thread: main: 18432
Thread: ForkJoinPool.commonPool-worker-3: 19456
Thread: ForkJoinPool.commonPool-worker-4: 3072

// Ausgabe 2:
Thread: ForkJoinPool.commonPool-worker-7: 2048
Thread: ForkJoinPool.commonPool-worker-1: 3072
Thread: ForkJoinPool.commonPool-worker-2: 5120
Thread: main: 22528
Thread: ForkJoinPool.commonPool-worker-5: 28320
Thread: ForkJoinPool.commonPool-worker-6: 10240
Thread: ForkJoinPool.commonPool-worker-3: 5120
Thread: ForkJoinPool.commonPool-worker-4: 23552
```

Hier werden einhunderttausend Tasks ausgeführt und die Namen der ausführenden Threads werden in einer Map gespeichert, wobei wir die Zuordnung der jeweiligen Aufgaben mitzählen. Da die Teilaufgaben zu klein sind (Jeder Task besteht lediglich darin den Namen des Threads in die Map zu speichern) werden für die Tasks je nach Ausführung mehr oder weniger Threads des Pools verwendet. So haben wir in der ersten Ausführung nur 5 der 8 Threads in Verwendung. Auch die Aufteilung, welcher Thread wieviele Aufgaben ausführt, ist hier sehr willkürlich gewählt. Sind die Teilaufgaben rechenintensiver so ist die Aufteilung auf die Threads gleichmäßiger.

Da bei parallelen Streams immer der Common-Pool verwendet wird, kann es bei der Ausführung mehrerer paralleler Streams innerhalb eines Systems zu Problemen kommen. Wenn sehr rechenintensive, beziehungsweise blockierende Tasks bei parallelen Streams verwendet werden, kann es vorkommen, dass der zweite parallele Stream quasi sequenziell abläuft, da alle Threads des Common-Pools von einem anderen parallelen Stream in Verwendung sind. Sequenziell deshalb weil dann nur noch der aufrufende Thread den parallelen Stream abarbeiten kann. Man kann dem parallelen Stream auch nicht direkt einen anderen Thread-Pool zuweisen. Es gibt jedoch zwei Tricks um dieses Problem zu umgehen. Die erste Möglichkeit besteht darin die Parallelität, also die Anzahl der im Common-Pool vorhandenen Threads über eine Systemeigenschaft von Java zu erhöhen.

```
System.setProperty(
  "java.util.concurrent.ForkJoinPool.common.parallelism", "16");
```

So kann man die Anzahl zum Beispiel auf sechzehn erhöhen. Dadurch hat man eine höhere Wahrscheinlichkeit, dass Threads für die Ausführung der parallelen Streams zur Verfügung stehen. Wenn die Erhöhung auch keine Verbesserung bringt, so kann man mit Hilfe folgenden Codes dem parallelen Stream einen anderen ForkJoinPool zuweisen, obwohl dies nicht direkt bei parallelen Streams ermöglicht wird.

```
ForkJoinPool forkJoinPool = new ForkJoinPool(100);
forkJoinPool.submit(() -> PARALLELER STREAM).get();
```

Hier erzeugen wir einen neuen ForkJoinPool mit einhundert Threads. Über submit können wir den parallelen Stream als *Runnable* übergeben. Über *get* wird die Ausführung gestartet. Nun werden zur Verarbeitung des Streams die Threads aus unserem ForkJoinPool anstelle des CommonPools verwendet. Interessanterweise wird, sofern man statt *get* die Methode *invoke* verwendet, eine Mischung aus beiden ForkJoinPools verwendet. Man sollte daher eher die Systemeigenschaft setzen und sich nicht auf einen eigenen ForkJoinPool verlassen.[Orab]

4.7 Interner Aufbau von Streams

In diesem Kapitel wollen wir uns den internen Aufbau von Streams in Java genauer ansehen. Dazu werden wir zuerst den Aufbau von Spliteratoren mit jenem von Iteratoren vergleichen. Anschließend werden wir den inneren Aufbau von sequenziellen und parallelen Streams und deren Verwendung des Fork-Join-Frameworks anhand eines Beispiels veranschaulichen.

4.7.1 Iteratoren und Spliteratoren

Über einen Iterator erhält man Zugriff auf die Elemente einer Aggregation von Daten. Der Zugriff auf die Daten erfolgt sequenziell und legt dabei nicht die innere Struktur der verwendeten Collection offen. Es handelt sich bei einem Iterator um ein Entwurfsmuster, welches häufig in objektorientierten Programmiersprachen Anwendung findet. Das Durchlaufen der Daten muss dabei nicht von der Collection selbst gehandhabt werden. Diese Aufgabe wird an den Iterator übergeben. Dadurch müssen auch keine weiteren Methoden in den Collections für das Iterieren der Daten vorhanden sein. Des Weiteren können auch mehrere Iteratoren ein und die selbe Collection durchlaufen. Je nach Aufbau der Datenstruktur gibt es verschiedene Möglichkeiten deren Daten zu iterieren. Der Vorteil eines Iterators besteht hier darin, dass man für das Durchlaufen einer komplexeren Datenstruktur einfach einen anderen Iterator verwenden kann um dies zu realisieren. Außerdem können robuste Iteratoren mit einer Veränderung der Kollektion während der Iteration umgehen. Hingegen würde beispielsweise beim Durchlaufen einer Liste über eine for-each Schleife eine *ConcurrentModificationException* beim Löschen eines Datensatzes geworfen werden.[GHJV95]

```
public Interface Iterator<E> {
    boolean hasNext();
    E next();
    default void remove();
    default void forEachRemaining(Consumer<? super E> action)
}
```

In Java wurde das Interface Iterator mit Version 1.2 hinzugefügt. Davor wurde in Java das Interface Enumeration für die Iteration verwendet. In Java 5 erhielt der Iterator zusätzlich einen Typparameter. In Java 8 wurden *remove* und *forEachRemaining* als Default-Methoden hinzugefügt. Die relevanten Methoden für die Funktionsweise eines Iterators sind *hasNext* um zu überprüfen ob weitere Elemente für das Durchlaufen verfügbar sind und *next*, wodurch das nächste Element der Collection zurückgegeben wird. Als Unterklasse kann für Listen das Interface *ListIterator* verwendet werden. Dadurch können auch vorherige Elemente iteriert werden. Außerdem können auch während der Iteration neue Elemente hinzugefügt werden. Es handelt sich also um einen robusten Iterator.

Man könnte nun auf die Idee kommen den Iterator auch für parallele Abarbeitung von Kollektion zu verwenden. Dazu würde man beispielsweise einen Iterator für eine Liste über *list.iterator()* erzeugen und anschließend an mehrere Threads oder über mehrere Tasks an einen Thread-Pool zur Ausführung übergeben. Das Problem bei diesem Ansatz ist jedoch, dass der Iterator nicht threadsicher ist. Es handelt sich bei der Abfrage von *hasNext* und *next* nicht um eine einzige atomare Methode. So könnte es passieren, dass bei Thread A für hasNext der Wert true zurück geliefert wird und während sich Thread B das nächste Element holen will, wurde es bereits von Thread A geholt und hasNext hätte somit, sofern der Iterator threadsicher wäre, false zurückliefern müssen. Um das Problem der parallelen Programmierung im Zusammenhang mit Collections zu lösen, könnte man beispielsweise aus einer Liste manuell mehrere Sub-Listen anfertigen und diese dann den jeweiligen Tasks oder Threads zuweisen (wie beim Fork-Join-Framework üblich). Eine andere Möglichkeit wäre es eine synchronisierte und dadurch threadsichere Warteschlange zu verwenden. Bei diesen Lösungsansätzen hat man jedoch das Problem, dass es einen zusätzlichen Aufwand für den Programmierer bedeuten würde. Zusätzlich würde man die eigentliche Logik des Codes mit zusätzlichem Code für parallele Ausführung vermischen. Außerdem könnte man hierbei Performance-Einbuße davontragen.[GS11]

Als Lösung von parallelen Iteratoren kommen in Java so genannte Spliteratoren zum Einsatz. Betrachten wir zunächst den Aufbau in Java.[Orab]

```
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<? super T> action);

    default void forEachRemaining(Consumer<? super T> action) {
        do { } while (tryAdvance(action));
    }

    Spliterator<T> trySplit();

    long estimateSize();
    default long getExactSizeIfKnown()
```

```
    int characteristics();
    default boolean hasCharacteristics(int characteristics);

    // weitere Methoden für primitive Spliteratoren
}
```

Spliteratoren kommen wie Iteratoren zum Durchlaufen einer Collection zum Einsatz und sind als Interface definiert. Die eigentliche Implementierung hängt von der jeweiligen Collection ab. Zusätzlich zu den Eigenschaften des Iterators kann der Spliterator die zu durchlaufende Collection aufsplitten. Dadurch können Spliteratoren in Zusammenarbeit mit parallelen Streams und mit Hilfe des verwendeten ForkJoinPools über eine Daten-Parallelität eine Task-Parallelität umsetzen. Daten-Parallelität wird hier durch die Spliteratoren erreicht, da sie die Daten einer Collection für eine parallele Abarbeitung in mehrere Teile aufteilen können. Über den ForkJoinPool können wir dann eine Task-Parallelität für Performance-Verbesserungen ausnutzen, da hier die davor aufgeteilten Daten auf mehrere Tasks aufgeteilt werden.[SSOG93]

Die aufgeteilten Collections können anschließend als Task für den ForkJoinPool parallel abgearbeitet werden. Des Weiteren ist die Verwendung eines Spliterators effizienter als die eines Iterators. Die Methoden *hasNext* und *next* wurden hier nämlich zusammengefasst zur Methode *tryAdvance* des Spliterators. Dabei wird, sofern weitere Elemente vorhanden sind, die übergebene Aktion in Form eines *Consumers* ausgeführt. Die Methode gibt dabei den Wert *true* zurück. Sind keine weiteren Elemente in der Collection, so wird false zurückgeliefert. Die Methode *forEachRemaining* führt für jedes verbleibende Element die übergebene Aktion sequenziell aus. Bei *trySplit* geht es darum, die Daten der Collection auf einen weiteren Spliterator aufzuteilen. So wird beispielsweise die erste Hälfte einer Liste vom aktuellen Spliterator durchlaufen. Die zweite Hälfte wird an den von *trySplit* zurückgegebenen Spliterator übernommen. Wenn sich das Aufteilen der Daten nicht mehr auszahlt oder nicht mehr möglich ist, so liefert diese Methode null zurück. Dadurch weiß man, dass die restlichen Daten sequenziell abgearbeitet werden sollen. Je genauer die Collections dabei in gleich große Teile zerteilt werden können, umso besser kann die Parallelität der Daten ausgenutzt werden. Dazu werden die Methoden *estimateSize* und *getExactSizeIfKnown* verwendet. Wenn man einen Spliterator lediglich für sequenzielle Streams verwenden will, so kann man bei *trySplit()* null zurückgeben. Dadurch wird nur ein einziger Spliterator für die gesamte Stream-Quelle verwendet.

Jeder Spliterator kann eine oder mehrere Eigenschaften besitzen. Diese werden bitweise mit Oder verknüft. Dadurch erhält man einen Wert vom Typ *int* zurück. Über die Methode *hasCharacteristics* kann auf eine oder mehrere Eigenschaften geprüft werden. Spliteratoren sind wie Iteratoren nicht Thread-Safe. Daher sollte zu jeder Zeit nur ein Thread auf einen Spliterator zugreifen. Parallele Berechnungen werden deshalb ermöglicht, weil über *trySplit* neue Spliteratoren erzeugt werden. Ein Spliterator kann die folgenden Eigenschaften besitzen: ORDERED, DISTINCT, SORTED, SIZED, NONNULL, IMMUTABLE, CONCURRENT, SUBSIZED. Inwiefern gewisse Eigenschaften die Performance

bei paralleler Abarbeitung beeinflussen können werden wir im Performance Kapitel klären.[Orab]

4.7.2 Stream-Umsetzung in Java

Nachdem wir uns die Grundlagen für sequenzielle und parallele Streams genauer angeschaut haben, werden wir in diesem Unterkapitel den internen Aufbau von Streams in Java genauer untersuchen. Mit Hilfe des folgenden, relativ einfachen Beispiels werden wir uns den inneren Aufbau von Streams ansehen. Zuerst betrachten wir den sequenziellen Stream, anschließend den Aufbau von parallelen Streams.[Orab]

Beispiel: Ablauf Sequenzieller Stream

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

list.stream()
    .filter(x -> x % 2 == 0)
    .map(x -> x + 100)
    .forEach(x -> System.out.println(x));
```

Mit *list.stream* wird der Stream erzeugt. Dabei passiert in Java folgendes. Über die Default-Methode der Collection wird über die Hilfsklasse *StreamSupport* der Stream vom Typ Integer erzeugt.

```
Collections.java:

default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

Der Methode *Streamsupport.stream()* wird dabei ein neu erzeugter Spliterator übergeben. Dafür würde es in *Collection* eine Methode geben. Ein effizienterer Spliterator für die hier verwendete Liste wird über die überschriebene Methode *spliterator()* von *ArrayList* bereitgestellt. Genauer gesagt liefert diese einen *ArrayListSpliterator* zurück. Man sieht hier, dass auch sequenzielle Streams Spliteratoren verwenden, da diese einen geringeren Overhead als Iteratoren haben und daher performanter sind. Der zweite Übergabeparameter *false* der Methode bestimmt ob es sich um einen parallelen Stream handelt. Die Klasse *StreamSupport* bietet mehrere Methoden zur Erstellung von sequenziellen und parallelen Streams an.

StreamSupport.java:

```
public static <T> Stream<T> stream(Spliterator<T> spliterator,
                                   boolean parallel) {
    Objects.requireNonNull(spliterator);
    return new ReferencePipeline.Head<>(
        spliterator,
        StreamOpFlag.fromCharacteristics(spliterator),
        parallel);
}
```

Hier wird über den zuvor erzeugten Spliterator ein neues *ReferencePipeline.Head* Objekt erzeugt. Diese Pipeline stellt die Operationen dar, welche auf die Daten des Streams angewandt werden sollen. Als *Head* wird hier die Quelle des Streams angesehen. In dieser *ReferencePipeline* sind auch die Methoden für *filter*, *map* und sonstige Zwischenoperationen definiert. Wie wir bereits im Verlauf dieser Arbeit gehört haben, liefern Zwischenoperationen ebenfalls Streams zurück. Genauer gesagt werden hier *StatelessOp*, also zustandslose Operationen, als Unterklasse der *ReferencePipeline* zurückgeliefert. Dort wird über die Methode *opWrapSink* ein verketteter *Sink* zurückgegeben.

Ein *Sink* ist dabei eine spezielle Erweiterung des Consumer-Interfaces. Dieser wird dazu verwendet, um die Werte des Streams durch die einzelnen Pipeline-Stages zu führen. So wird über *accept* der jeweilige Wert zur nächsten Stufe der Pipeline weitergegeben. Dabei bezeichnet man die nächste Stufe als *downstream*. Bevor *accept* aufgerufen werden darf, muss vorher die Methode *begin()* und abschließend *end()* aufgerufen werden.

ReferencePipeline.java:

```
// filter Methode -> Sink.accept
public void accept(P_OUT u) {
    if (predicate.test(u))
        downstream.accept(u);
}
```

So sieht beispielsweise der relevante Teil der Implementierung der Methode *filter* aus. Die Werte werden nur an die nächste Pipeline-Stufe *downstream* geschickt sofern der übergebene Wert dem Prädikat entspricht. Wenn die Pipeline aufbauend auf den unterschiedlichen Zwischen-Operationen fertig zusammengestellt wurde, wird die Berechnung über die terminierende Operation in Gang gesetzt. Wie wir bereits erfahren haben, wird diese Eigenschaft als *lazy evaluation* bezeichnet. In der ReferencePipeline wird nun *forEach* aufgerufen.

```
ReferencePipeline.java:

public void forEach(Consumer<? super P_OUT> action) {
    evaluate(ForEachOps.makeRef(action, false));
}

AbstractPipeline.java:
final <R> R evaluate(TerminalOp<E_OUT, R> terminalOp) {
    ...
    return isParallel()
      ? terminalOp.evaluateParallel
        (this, sourceSpliterator(terminalOp.getOpFlags()))
      : terminalOp.evaluateSequential
        (this, sourceSpliterator(terminalOp.getOpFlags()));
}
```

Über *ForEachOps.makeRef* wird eine Instanz eines Terminal-Operators, welcher das Interface *TerminalOp* implementiert, erzeugt. Die Methode *evaluate* wird in der Klasse *AbstractPipeline* implementiert und ruft, je nachdem ob es sich um einen sequenziellen oder parallelen Stream handelt, *evaluateSequential* beziehungsweise *evaluateParallel* der terminierenden Operation auf.

```
AbstractPipeline.java:

final <P_IN> void copyInto(Sink<P_IN> wrappedSink,
                           Spliterator<P_IN> spliterator) {
    ...
    spliterator.forEachRemaining(wrappedSink);
    ...
}
```

Über *copyInto* wird die Verarbeitung letztlich angestoßen. Dabei werden dem Anfangs erzeugten Spliterator die Pipeline-Operationen in Form eines verpackten *Sink* übergeben. Über die Methode *forEachRemaining* werden die Elemente des ursprünglichen Streams sequenziell durch die Pipeline geschickt. In diesem Beispiel erhalten wir nun folgende Ausgabe: 102, 104, 106, 108, 110. Nun betrachten wir dasselbe Beispiel mit einem parallelen Stream.

Beispiel: Ablauf Paralleler Stream

Der parallele Stream verwendet genau denselben Spliterator wie der sequenzielle Stream. Auch die ersten Schritte bei denen die einzelnen Operationen der Pipeline aufgebaut werden sind gleich. Daher steigen wir bei diesem Beispiel erst bei der Methode *evaluate*

der Klasse *AbstractPipeline* ein. Hier wird nun *terminalOp.evaluateParallel* aufgerufen. In dieser Methode wird ein *ForEachTask* erzeugt. Dabei handelt es sich um eine Erweiterung der Klasse *CountedCompleter*. In den anderen Klassen für Operationen innerhalb von *java.util.stream*, werden basierend auf der abstrakten Klasse *AbstractTask*, sonstige Tasks für den ForkJoinPool für die parallele Abarbeitung bereitgestellt.

Beim *CountedCompleter* handelt es sich um einen ForkJoinTask, welcher im vorherigen Kapitel noch nicht vorgestellt wurde. Der Unterschied zu den anderen beiden ForkJoinTasks ist, dass hier mitgezählt wird, wie viele SubTasks durch diesen Task erzeugt wurden. Wenn alle SubTasks ausgeführt wurden, kann mit Hilfe des *CountedCompleter* ein abschließender Task ausgeführt werden. Bei unserem Beispiel mit *forEach* als terminierende Operation ist diese Zusatzfunktion nicht relevant. Sie wird aber dann benötigt, wenn Ergebnisse von Sub-Tasks zusammengeführt werden müssen, wie beispielsweise bei der *reduce*. Wenn alle Sub-Tasks ein Ergebnis berechnet haben, so kann man mit *tryComplete()* alle Teilergebnisse zusammenzählen und an den darüberliegenden Task weitergeben. Dadurch gelangt man dann zum Endergebnis des Problems.

Beim Aufruf der Methode *terminalOp.evaluateParallel* werden also die Tasks für das Fork-Join-Framework erzeugt. Jedem *ForEachTask* werden dabei ein Spliterator, ein *Sink* und der *Pipeline-Helper* übergeben. In der im Task überschriebenen Methode *compute* werden mit Hilfe des Spliterators über *trySplit* neue Spliteratoren erzeugt. Dadurch werden dann so lange neue Tasks erzeugt, bis die zu lösenden Probleme klein genug sind. Das geschieht wenn *trySplit* null zurückliefert. Für jeden so erzeugten Spliterator wird ein neuer Task angelegt und geforkt. Die Worker-Threads des ForkJoinPools können dann wie beim sequenziellen Stream mittels *forEachRemaining* die jeweiligen Sub-Listen abarbeiten. Wann die Probleme klein genug sind um sie sequenziell abzuarbeiten, wird dabei in der Methode *compute()* bestimmt. Es wird dabei nur so lange *trySplit* aufgerufen, so lange die geschätzte Größe der Sub-Liste größer als ein davor berechneter Grenzwert ist.

```
while (sizeEstimate > sizeThreshold &&
        (ls = rs.trySplit()) != null) {
    // erzeuge neue Tasks
}
```

Gleichzeitig muss aber auch trySplit des Spliterators einen neuen Spliterator zurückliefern. Den Wert von *sizeEstimate* erhalten wir vom Spliterator. Der Grenzwert *sizeThreshold* wird folgendermaßen berechnet.

```
static final int LEAF_TARGET =
    ForkJoinPool.getCommonPoolParallelism() << 2;

public static long suggestTargetSize(long sizeEstimate) {
    long est = sizeEstimate / LEAF_TARGET;
```

```
    return est > 0L ? est : 1L;
}
```

Da wir eine CPU mit acht Kernen verwenden, *getCommonPoolParallelism* aber wie erwähnt einen weniger zurückliefert und derzeit für parallele Streams die vierfache Anzahl an Tasks vorgegeben wird, erhalten wir für unser Testsystem für *LEAF_TARGET* den Wert 28 (Der left-Shift mit Wert zwei entspricht einer Multiplikation mit vier). Wenn wir nun einen Stream mit einer Million Elementen parallel Verarbeitung wollen, so werden also 28 Tasks erzeugt, wobei die Grenze bei 35.714 Elementen liegt.

4.8 Vergleich zwischen Streams und Schleifen

Nachdem wir nun die Streams API kennen gelernt haben, wollen wir in diesem Unterkapitel die Unterschiede zwischen Streams und for-Schleifen genauer betrachten. Wir wollen außerdem die Frage klären, ob es sich immer auszahlt Streams anstelle von for- beziehungsweise for-each Schleifen zu verwenden. Diese Frage werden wir über mehrere Beispiele beantworten. Als Domäne für die Beispiele verwenden wir ein imaginäres Verwaltungssystem für Studenten. Dabei geht es um die Verwaltung von Studenten und Lehrveranstaltungen. Wir fangen zunächst mit einfachen Beispielen an. Wir versuchen zuerst alle Studenten zu finden, die einen Wohnsitz in Wien haben. Diese wollen wir per Matrikelnummer ausgeben. Bei den folgenden Beispielen gehen wir davon aus, dass alle Felder einer Klasse einen Wert besitzen und ersparen uns dadurch die Überprüfung auf null.

```
studenten.stream()
  .filter(student -> student.getAdresse()
                            .getStadt().equals("Wien"))
  .map(Student::getMatrikelnummer)
  .forEach(System.out::println);

for(Student student : studenten) {
  if(student.getAdresse().getStadt().equals("Wien")) {
    String matrikelnummer = student.getMatrikelnummer();
    System.out.println(matrikelnummer);
  }
}
```

Bei diesem einfachen Beispiel erkennt man bereits den Vorteil von Streams, auch wenn hier die Länge des Codes bei beiden Varianten in etwa gleich lang ist. Die Problemstellung ist hier auf Grund der namhaften Bezeichnungen der Stream-Operationen besser herauszulesen als bei der Schleife. Durchlaufe alle Studenten, filtere sie nach dem Wohnsitz *Wien* und gib deren Matrikelnummer aus. Wie bereits in dieser Arbeit erwähnt, sagen

wir hier nicht wie es gemacht werden soll. Wir geben über *filter*, *map* und *forEach* nur an was gemacht werden soll. Dadurch ist die Variante mit dem Stream einerseits aussagekräftiger und andererseits müssen wir uns nicht um interne Implementierungen kümmern. Betrachten wir nun weitere Beispiele bei denen diese Vorteile noch deutlicher zum Vorschein kommen.

Im nächsten Beispiel wollen wir uns die zehn besten Studenten aus Wien zurückgeben lassen. Diese wollen wir als Liste erhalten. Die Studenten sollen dabei nach Notendurchschnitt sortiert werden.

```
return studenten.stream()
  .filter(s -> s.getAdresse().getStadt().equals("Wien"))
  .sorted(Comparator.comparing(Student::getNotendurchschnitt))
  .limit(10)
  .collect(Collectors.toList());
```

Hier kann man wieder problemlos herauslesen was gemacht wird. Die Studenten werden zuerst nach Wohnort gefiltert, anschließend sortiert und die ersten zehn Ergebnisse werden als Liste zurückgegeben. Dies könnten wir noch etwas vereinfachen indem wir Collectors.toList() über folgenden statischen Import importieren *import static java.util.stream.Collectors.toList*. Dadurch muss man nur mehr *.collect(toList())* schreiben. Selbes können wir auch für *Comparator.comparing* machen. Schauen wir uns als nächstes eine Lösung mit Schleifen und ohne Lambda-Ausdrücke an.

```
studenten.sort(new Comparator<Student>() {
    @Override
    public int compare(Student o1, Student o2) {
        return Double.compare(o1.getNotendurchschnitt(),
            o2.getNotendurchschnitt());
    }
});

List<Student> result = new ArrayList<>();
int counter = 0;
for(Student student : studenten) {
    if(student.getAdresse().getStadt().equals("Wien")) {
        result.add(student);
        counter++;
        if(counter==10) {
            break;
        }
    }
}
return result;
```

Hier gibt es mehrere Nachteile. Zum einen ist hier nicht klar erkenntlich welches Problem wir eigentlich lösen wollen. Wir sortieren zuerst alle Studenten nach Notendurchschnitt. Dies ist jedoch unleserlich über eine anonyme innere Klasse umgesetzt. Zusätzlich verwenden wir *Double.compare* um nach Notendurchschnitt zu sortieren. Dadurch wird die Collection *studenten* verändert, da die Methode sort die Datenquelle an sich manipuliert. Beim Stream haben wir dieses Problem nicht. Die darunterliegende Quelle wird nicht verändert. Das bedeutet, sofern wir bei der Umsetzung mit anonymen inneren Klassen die Collection nicht bearbeiten wollen, müssen wir vorher noch eine Kopie dieser erzeugen und erst anschließend *sort* ausführen. Zusätzlich müssen wir explizit eine neue Liste für die Ergebnisse anlegen. Außerdem müssen wir entweder einen Zähler verwenden, um aus der Schleife herauszuspringen, wenn wir genügend Studenten zur Ergebnisliste hinzugefügt haben. Oder aber wir kontrollieren direkt nach dem Hinzufügen von Elementen die Größe der Liste.

Die Lösung mit Streams ist nicht nur lesbarer, sondern kann auch leichter gewartet werden. Wir können über *sorted comparing* viel einfacher nach einem neuen Attribut sortieren lassen. Außerdem können wir hier, sofern wir dem Benutzer/der Benutzerin unserer Methode mehr Freiheiten geben wollen, einfach den Stream zurückgeben. Dadurch kann man noch zusätzliche Stream-Operationen durchführen und anschließend selbst bestimmen, in welcher Form man die Daten mittels *collect* oder *reduce* erhalten möchte.

Abschließend wollen wir uns noch ein komplexeres Beispiel anschauen. Dabei wollen wir sehen, ob auch verschachtelte Schleifen lesbar als Streams dargestellt werden können. Hier das Beispiel, das wir umsetzen möchten: Ein Professor leitet mehrere Lehrveranstaltungen. Jede dieser Lehrveranstaltungen hat mehrere Studierende als Teilnehmer. Wir wollen für einen Professor alle teilnehmenden Studierenden ermittelt. Dabei sollen keine Duplikate vorkommen. Außerdem wollen wir die Studenten nach Geschlecht gruppieren und anschließend nach Herkunftsland. Das heißt, wir wollen hier eine Multi-Map als Ergebnis, dessen Signatur wie folgt aussieht: Map<String, Map<String, List<Student≫. Diese Multi-Map könnte man anschließend zum Erstellen von Statistiken verwenden. Nachfolgend sehen wir das gelöste Beispiel mittels Stream.

```
Map<String, Map<String, List<Student>>> multimap =
  professor.getLvas()
    .stream()
    .flatMap(lva -> lva.getStudenten().stream())
    .distinct()
    .collect(groupingBy(Student::getGeschlecht,
             groupingBy(Student::getStaatsbuergerschaft)));
```

Über *flatmap* und *distinct* bekommen wir alle Studenten der Lehrveranstaltungen ohne Duplikate. Die Operation *groupingBy* innerhalb von *collect* können wir verketten um unsere Multi-Map zu erzeugen. Sofern man sich nicht all zu sehr mit der Streams API beschäftigt hat, könnte man hier schon Probleme beim Lesen des Codes bekommen. Vor

allem wenn man davor eher weniger mit funktionaler Programmierung zu tun hatte. Dennoch sieht die Problemlösung sehr kompakt aus. Betrachten wir nun die Lösung mittels for-each-Schleife.

```
Map<String, Map<String, List<Student>>> multimap =
    new HashMap<>();
Set<Student> distinctStudenten = new HashSet<>();

for (Lva lva : professor.getLvas()) {
  for (Student student : lva.getStudenten()) {
    distinctStudenten.add(student);
  }
}

for (Student student : distinctStudenten) {
  if (multimap.get(student.getGeschlecht()) == null) {
    multimap.put(student.getGeschlecht(), new HashMap<>());
  }

  Map<String, List<Student>> map =
      multimap.get(student.getGeschlecht());

  if (map.get(student.getStaatsbuergerschaft()) == null) {
    map.put(student.getStaatsbuergerschaft(), new ArrayList<>());
  }

  List<Student> studenten =
      map.get(student.getStaatsbuergerschaft());
  studenten.add(student);
}
```

Um nur unterschiedliche Studenten zu bekommen, müssen wir zuerst die Operation *distinct* nachbauen. Dies geschieht hier über ein *Set*, da es keine Werte doppelt beinhalten kann. Anschließend durchlaufen wir das soeben erzeugte Set erneut mittels Schleife. Hier müssen wir nun unsere Multimap selbst zusammenbauen. Daher wird zuerst überprüft, ob die äußere Map bereits als Schlüssel das jeweilige Geschlecht beinhaltet. Andernfalls erzeugen wir einen neuen Eintrag, bestehend aus dem Geschlecht als String und einer neuen HashMap mit Signatur *HashMap<String, List<Student≫* als dazugehörigen Wert. Danach gibt es die selbe Überprüfung mit der Staatsbürgerschaft. Sofern ein Land noch nicht eingetragen wurde, erzeugen wir einen neuen Eintrag in der inneren HashMap und befüllen diesen mit dem Namen des Landes als Schlüssel und einer neuen Liste als Wert. Danach holen wir uns immer die geeignete Liste, in der wir den Studenten hinzufügen

können. Hier sieht man wieder, dass die Variante mit den Streams einfacher aufgebaut ist und uns einiges an zusätzlichem Code und Aufwand für die Logik dahinter erspart.

4.8.1 Parallele Verarbeitung der Daten

Hier wollen wir anhand eines weiteren Beispiels betrachten, wie hoch der Aufwand ist, sequenzielle Lösungen in parallele Lösungen weiterzuentwickeln. Dazu setzen wir folgendes Beispiel um: Wir wollen für alle Studenten eine Statistik berechnen lassen. Dazu verwenden wir die Liste aller Studenten und rufen anschließend die Methode für die Berechnung der Statistik auf. Anschließend geben wir die dadurch erzeugten Objekte der Klasse *Statistik* als Liste zurück. Wir gehen nun davon aus, dass die Berechnung ein komplexer Vorgang ist und wollen diese daher parallel ausführen.

```
// Berechnen der Statistik mit Stream
studenten.stream()
         .map(Student::berechneStatistik)
         .collect(Collectors.toList());

// Berechnen der Statistik mit Schleife
List<Statistik> statistiken = new ArrayList<>();

for(Student student : studenten) {
    Statistik statistik = student.berechneStatistik();
    statistiken.add(statistik);
}
```

Schauen wir uns zunächst an, was wir am Code ändern müssen, um die Lösung mittels Schleife parallel ablaufen zu lassen. Dazu verwenden wir das im vorherigen Kapitel vorgestellte Fork-Join-Framework und erzeugen eine neue Klasse, welche *RecursiveTask* erweitert.

```
public class StatistikRecursiveTask
      extends RecursiveTask<List<Statistik>> {
  private static final int GRENZE = 1;
  private List<Student> studenten;
  private int von;
  private int bis;

  public StatistikRecursiveTask(List<Student> studenten) {
    this.studenten = studenten;
    this.von = 0;
    this.bis = studenten.size();
  }
```

```
  private StatistikRecursiveTask(List<Student> studenten,
                                 int von, int bis) {
    this.studenten = studenten;
    this.von = von;
    this.bis = bis;
  }

  @Override
  protected List<Statistik> compute() {

  List<Statistik> statistiken = new ArrayList<>();

  int subsize = bis - von;

  if (subsize <= GRENZE) {
    for (int i = von; i < bis; i++) {
      Statistik statistik =studenten.get(i)
                                    .berechneStatistik();
      statistiken.add(statistik);
    }
  } else {
    // teile probleme in kleinere teile auf
    int indexTeilung = (bis - von) / 2 + von;

    StatistikRecursiveTask links =
        new StatistikRecursiveTask(studenten, von, indexTeilung);
    StatistikRecursiveTask rechts =
        new StatistikRecursiveTask(studenten, indexTeilung, bis);

    links.fork();
    rechts.fork();

    List<Statistik> resultatLinks = links.join();
    List<Statistik> resultatRechts = rechts.join();

    statistiken.addAll(resultatLinks);
    statistiken.addAll(resultatRechts);
  }

  return statistiken;
}
```

Die Berechnung starten wir folgendermaßen: Wir erzeugen einen neuen *StatistikRecursiveTask* und stoßen die Berechnung mit *invoke()* an. Wir erhalten die Liste der Statistiken als Rückgabewert.

```
StatistikRecursiveTask task =
    new StatistikRecursiveTask(studenten);
List<Statistik> resultat = task.invoke();
```

Am Anfang wird die Liste der Studenten an den Task übergeben. Zusätzlich dazu müssen wir uns für jeden Task merken, für welche Bereiche der Liste er zuständig ist. Dazu verwenden wir die beiden Werte *von* und *bis*. Da wir auf die Listen nur lesend zugreifen, verwendet jeder Subtask dieselbe Liste von Studenten. Man könnte es natürlich auch über eigene Sublisten lösen. Beim Erstellen des Tasks legen wir die Indizes von null bis zum Ende der Liste an. In der überschriebenen Methode *compute* passiert Folgendes: *subsize* bezeichnet hier die Länge der Subliste um die sich der Task kümmert. Nun gibt es zwei Möglichkeiten. Wenn die Größe der simulierten Subliste kleiner gleich einer vorgegebenen Größe ist, so können wir die Berechnung sequenziell durchführen. Andernfalls brechen wir das Problem in kleinere Teilprobleme runter. Die Teilung erfolgt dadurch, dass wir die Mitte der derzeitigen Subliste ermitteln und anschließend zwei neue Tasks dafür erzeugen. Über *fork()* starten wir die neu erzeugten Subtasks. Über *join()* warten wir auf die Ergebnisse der Subtasks und anschließend fügen wir die beiden Ergebnisse über unsere Resultat-Liste *statistiken* zusammen.

Schauen wir uns nun die Änderung am Stream an um die parallele Berechnung dort ebenfalls durchzuführen.

```
List<Statistik> resultatStream =
    studenten.parallelStream()
             .map(Student::berechneStatistik)
             .collect(Collectors.toList());
```

Hier ändern wir lediglich *.stream()* auf *.parallelStream()*. Hier sehen wir einen entscheidenden Vorteil im Gegensatz zu Schleifen. Will man bei Schleifen auf eine parallele Ausführung wechseln, so muss man sich mit dem Fork-Join-Framework auseinandersetzen. Man muss eine eigene Klasse erzeugen und sich überlegen wie die *compute* Methode aussehen könnte. Anschließend muss man sich überlegen, wie man die Probleme in kleinere Teilprobleme aufsplitten kann. Übergibt man mehrere Sublisten oder kann man immer dieselbe Liste verwenden. Hier muss man auch auf Thread-Sicherheit achten. Solange man nur lesende Zugriffe macht, wie in unserem Beispiel, ist Thread-Sicherheit jedoch gegeben. Doch es gibt noch weitere Probleme bei jener Variante mit *compute*. Wenn wir uns die direkte Berechnung ansehen, so können wir erkennen, dass wir über die for-Schleife mittels *get* auf die Liste zugreifen. Dabei müssen wir darauf achten, dass wir in der Schleife die richtigen Indizes angeben. So könnte ein *kleiner gleich* statt dem $i < bis$ die Berechnungen

fehlerhaft machen, oder aber zu viele Statistiken zurückliefern. Auch bei der Erstellung der Teilprobleme, müssen wir die richtigen Indizes angeben, um ein korrektes Ergebnis erhalten zu können. Das heißt, wir müssen hier unsere Lösung ausgiebig Debuggen und Testen. Beim Stream sparen wir uns einiges an zusätzlichem Code. Hier ändern wir lediglich eine Methode. Der Rest geschieht im Hintergrund. Einziger Vorteil bei der Schleife ist, dass wir hier die Grenze selbst bestimmen können. Dadurch können wir die Anzahl erzeugter Tasks selbst steuern. und mittels Benchmarks feststellen, welcher Wert besser für Performance-Verbesserungen geeignet wäre.

4.8.2 Nachteile von Streams

Nach den bisher gezeigten Beispielen könnte man davon ausgehen, dass man immer Streams anstelle der herkömmlichen Schleifen verwenden sollte. Deshalb wollen wir in diesem Kapitel noch ein paar Nachteile von Streams aufzeigen.

Parallele Streams Wie wir gesehen haben, ist es bei Streams relativ einfach, von einer sequenziellen auf eine parallele Verarbeitung zu wechseln. Doch dadurch können auch Probleme entstehen. Man könnte auf die Idee kommen immer parallele Streams zu verwenden, da man denkt, dass mehr Kerne immer eine schnellere Verarbeitung der Daten bedeutet. Doch so einfach ist es bei parallelen Streams nicht. Des Weiteren kann, wie bereits erwähnt, die Korrektheit des Programms bei paralleler Ausführung beeinträchtigt werden, sofern wir beispielsweise bei *reduce* keine assoziativen Operationen verwenden. Auf die Problematik der parallelen Streams im Bezug auf die Performance werden wir im Kapitel *Performance* genauer eingehen.

Lesbarkeit Nicht immer sind Streams lesbarer als Schleifen. Gerade bei verschachtelten for-Schleifen kann es vorkommen, dass die Variante mit Schleifen lesbarer ist. Lesbarkeit spielt eine besondere Rolle in der Software-Entwicklung. Sie beeinflusst nämlich auch andere Faktoren, welche im Zusammenhang mit hoher Software-Qualität erwähnt werden. So erhöht eine gute Lesbarkeit auch die Wiederverwendbarkeit, Wartbarkeit und Zuverlässigkeit und verringert gleichzeitig die Komplexität des Codes.[TOAY13][Fit96]

```
IntStream.rangeClosed(1, 10)
  .forEach(x ->
      IntStream.rangeClosed(1, 10)
               .forEach(y ->
            IntStream.rangeClosed(1, 10)
                 .forEach(z ->
                     System.out.println(x*y*z))));
```

```
for (int x = 1; x <= 10; x++) {
  for (int y = 1; y <= 10; y++) {
    for (int z = 1; z <= 10; z++) {
      System.out.println(x * y * z);
    }
  }
}
```

Hier sieht man, dass man nicht immer aus einer Schleife einen Stream machen sollte. Da Lesbarkeit so einen wichtigen Bestandteil der Software-Entwicklung darstellt, sollte man hier doch eher die alte Variante mit Schleifen verwenden. In Hinblick auf die Lesbarkeit von Streams sollte man auch nicht zu komplexe Code-Blöcke als Lambda-Ausdrücke verwenden, sondern stattdessen neue Methoden erzeugen und anschließend Methoden-Referenzen innerhalb der Stream-Operationen verwenden.

Debugging Wenn während einer Stream-Operation ein Fehlerfall eintritt und dadurch eine Exception geworfen wird, so ist es auf Grund der Vorgänge die im Hintergrund eines Streams ausgelöst werden, nicht immer einfach den Fehler zu finden. Schauen wir uns dazu ein Beispiel an.

```
236: studenten.stream()
237:          .filter(Student::hatZeugnisse)
238:          .filter(Student::istFortgemeldet)
239:          .map(Student::berechneStatistik)
240:          .collect(toList());
```

Wenn nun beispielsweise bei *berechneStatistik* eine Exception geworfen wird, so erhalten wir folgende Fehlermeldung. Dabei wurden die Paketnamen auf Grund der Lesbarkeit entfernt.

```
Exception in thread "main" ArithmeticException: / by zero
  at Student.berechneStatistik(Student.java:80)
  at ReferencePipeline$3$1.accept(ReferencePipeline.java:193)
  at ReferencePipeline$2$1.accept(ReferencePipeline.java:175)
  at ReferencePipeline$2$1.accept(ReferencePipeline.java:175)
  at ArrayListSpliterator.forEachRemaining(ArrayList.java:1382)
  at AbstractPipeline.copyInto(AbstractPipeline.java:481)
  at AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:471)
  at ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)
  at AbstractPipeline.evaluate(AbstractPipeline.java:234)
  at ReferencePipeline.collect(ReferencePipeline.java:499)
  at Main.main(Main.java:241)
```

Hier sehen wir alle internen Aufrufe von *collect*, wobei die Exception immer nach oben hin weitergereicht wird, bis wir in unserer Klasse *Main* angekommen sind. Dadurch erhalten wir relativ viel unnötige Information bei dieser Fehlermeldung im Vergleich zur Variante mit Schleifen. Da die Berechnungen erst von der terminierenden Operation *collect* angestoßen werden, wird uns die Zeile mit *collect* als Verursacher der Exception angezeigt, obwohl der Fehler während dem Mapping geschieht. In diesem Fall sehen wir immerhin an der zweiten Zeile, dass das Problem an *berechneStatistik* liegt. Wenn wir jedoch anstelle von Methodenreferenzen Lambda-Ausdrücke verwenden, wird es schwer den Fehler auf Anhieb zu finden.

Auch beim manuellen Debuggen und Testen von Streams können Schwierigkeiten auftreten. So ist der genaue Ablauf innerhalb der Stream-Operationen nicht immer ohne zusätzliche Debug-Tools nachvollziehbar. Vorallem wenn man anstelle von Methoden-Referenzen Lambda-Ausdrücke innerhalb der Stream-Operationen verwendet. Mittlerweile kann man sich jedoch beispielsweise bei der integrierten Entwicklungsumgebung (IDE) *IntelliJ IDEA*[BRA] während des Debuggens eine grafische Darstellung des jeweiligen Streams anzeigen lassen.

```
IntStream.rangeClosed(0,10)
         .filter(x -> x % 2 == 0)
         .map(x -> x * 2)
         .forEach(System.out::println);
```

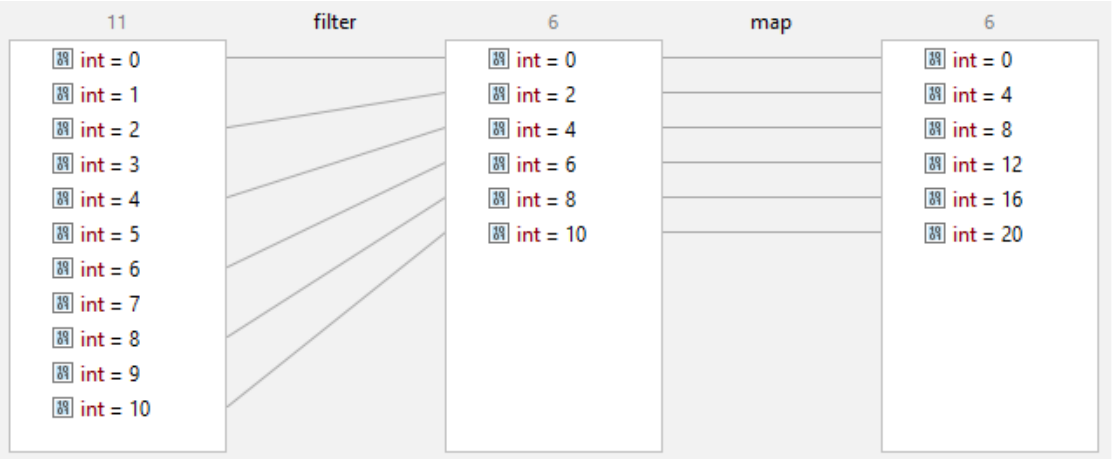


Abbildung 4.1: IntelliJ IDEA Stream Debugging[BRA]

In Abbildung 4.1 sehen wir die grafische Darstellung des davor beschriebenen *IntStream*. Dadurch kann man die Operationen besser nachverfolgen und man sieht auch, was mit den einzelnen Elementen des Streams geschieht. So kann man auch bei komplexeren Stream-Pipelines die Übersicht bewahren. Hat man diese Debugging-Tools nicht zur Verfügung, so

kann man mittels bereits vorgestellter Stream-Operation *peek* Log-Nachrichten zwischen den einzelnen Operationen hinzufügen.

Exception-Handling Kommen wir nun zum Exception-Handling innerhalb von Streams. Auch hier gibt es im Zusammenhang mit Lambda-Ausdrücken und der Streams API Probleme. In Java gibt es zwei Arten von Exceptions. Man unterscheidet zwischen *checked* und *unchecked* Exceptions. Checked Exceptions werden bei Methoden mittels *throws* angegeben. Dadurch weiß man, dass die Methode bei der Ausführung den angegebenen Fehler werfen kann und man sich um die Fehlerbehandlung kümmern muss. Das bedeutet, dass man entweder den Fehler abfängt oder ebenfalls mit einem *throws* an die ausführende Methode weiterleitet. Unchecked Exceptions hingegen müssen nicht angegeben werden. Diese werden im Gegensatz zu *checked* Exceptions nicht vom Compiler überprüft. Dazu gehören *Errors* und *RuntimeExceptions*[GJS+15]. Zu den *RuntimeExceptions*, also jenen Exceptions welche erst zur Laufzeit erkannt werden, gehört zum Beispiel die in Java bekannte *NullPointerException*. Diese tritt auf sofern auf eine nicht vorhandene Referenz zugegriffen wird.[RM00]

Zunächst betrachten wir *unchecked* Exceptions innerhalb von Streams. Dazu verwenden wir unser Debugging-Beispiel, bei dem wir innerhalb von *berechneStatistik* auf Grund einer Division durch 0 einen Fehler erhalten haben. Diesen wollen wir nun innerhalb des Streams mit *try/catch* abfangen.

```
studenten.stream()
         .filter(Student::hatZeugnisse)
         .filter(Student::istFortgemeldet)
         .map(Student::berechneStatistik)
         .collect(Collectors.toList());
```

Hier haben wir dank der Methoden-Referenz den Vorteil, dass wir innerhalb der Methode unser Exception-Handling vornehmen können, ohne dabei die Lesbarkeit des Streams zu beeinträchtigen. Wenn die Methode *berechneStatistik* beispielsweise aus einer Bibliothek oder einem Framework verwendet wird, so müssen wir uns direkt bei der Verwendung innerhalb des Streams um die Exception kümmern. Gleiches gilt auch für Lambda-Ausdrücke. Bei diesem Beispiel können wir bei *unchecked exceptions* entweder innerhalb der *map* Operation ein *try/catch* verwenden oder außerhalb des Streams. Sofern wir die Exception innerhalb behandeln hat dies Auswirkungen auf die Lesbarkeit des Streams.

Schauen wir uns nun *checked* Exceptions an. Hier müssen wir uns entweder um die Exception mittels *try/catch* kümmern oder wir werfen sie mit *throw* an den Aufrufer weiter.

```
// Nicht ausführbar
// Unhandled Exception: java.io.IOException
studenten.stream()
         .map(Student::erstelleDatei)
         .collect(Collectors.toList());
```

Hier gehen wir davon aus, dass die Methode *erstelleDate* innerhalb der Klasse *Student* eine *IOException* werfen kann. Daher müssen wir uns hier um die *checked* Exception kümmern. Leider haben wir bei Streams nicht direkt die Möglichkeit innerhalb der Stream-Operation ein *throws IOException* anzugeben. Dadurch können wir die Exception nicht außerhalb des Streams behandeln.

```
studenten.stream()
         .map(student -> {
             try {
                 return File.createTempFile("student_", "");
             } catch (IOException e) {
                 e.printStackTrace();
                 throw new RuntimeException();
             }})
         .collect(Collectors.toList());
```

Aus unserem kompakten Stream wird wegen *try/catch* ein schlecht lesbarer und unübersichtlicher Code-Block. In dieser Lösung werfen wir nun im *catch*-Block eine neue RuntimeException um die Berechnungen des Streams abzubrechen. Eine weitere Möglichkeit wäre den Lambda-Funktionskörper von *map* auszulagern und anschließend den Lambda-Ausdruck durch eine Methoden-Referenz zu ersetzen. Außerdem kann man einen so genannten *Wrapper* verwenden. Dazu müssen wir ein neues funktionales Interface definieren.

```
@FunctionalInterface
public interface CheckedFunction<T, R> {
  R apply(T t) throws Exception;
  static <T,R> Function<T,R> wrap(
        CheckedFunction<T,R> checkedFunction) {
    return x -> {
      try {
        return checkedFunction.apply(x);
      } catch (Exception e) {
        throw new RuntimeException(e);
      }
    };}}
```

Anschließend können wir die Methode *wrap* einsetzen. Dadurch wird die *checked* Exception ebenfalls in eine *unchecked* Exception umgewandelt.

```
studenten.stream()
         .map(CheckedFunction.wrap(Student::erstelleDatei))
         .collect(Collectors.toList());
```

Dabei müssen wir jedoch auch für andere funktionale Interfaces *Wrapper* erzeugen. Da lediglich der Java Compiler zwischen *checked* und *unchecked* Exceptions unterscheidet, nicht jedoch die JVM selbst[GJS+15], kann man mit einem Trick die Überprüfung des Compilers aushebeln. Dadurch wird es dann möglich *checked* Exceptions innerhalb von Streams wie unchecked Exceptions weiterzureichen. Dies sollte man jedoch vermeiden, da ein Aufrufer der jeweiligen Methode nicht damit rechnen würde, dass eine checked Exception von einer Methode geworfen wird, welche kein *throws* in der Methoden-Signatur deklariert. Die sichere Variante bleibt also, die checked Exception direkt abzufangen und als *Runtime-Exception* zu werfen.[Stab]

KAPITEL 5

Java Virtual Machine (JVM)

In diesem Kapitel liegt das Hauptaugenmerk auf der Umsetzung der Lambda-Ausdrücke innerhalb von Java 8. Dazu werden zuerst die Grundlagen der virtuellen Maschine vorgestellt. Anschließend werden wir den JVM-Befehl *invokedynamic*, welcher bereits in Java 7 für die Unterstützung von dynamisch typisierten Programmiersprachen in der JVM, hinzugefügt wurde, näher betrachten. In diesem Zusammenhang wurden auch Method-Handles hinzugefügt. Nachdem auch diese näher vorgestellt wurden, werden wir uns dem internen Aufbau von anonymen inneren Klassen widmen. Dadurch sollen die Unterschiede des internen Aufbaus von anonymen inneren Klassen zu jenen der Lambda-Ausdrücke in Java deutlich gemacht werden. Um einen Vergleich zu anderen JVM-Sprachen zu erhalten, werden wir anschließend die Umsetzung von Lambda-Ausdrücken in Scala betrachten.

5.1 JVM-Einführung

In diesem Unterkapitel werden wir die Funktionsweise der Java Virtual Machine (JVM) vorstellen. Für diese Einführung dient die *Java Virtual Machine Specification für Java 8* [LYBB15] als Hauptquelle. Es handelt sich bei der JVM um eine abstrakte Rechenmaschine. Sie verfügt über einen eigenen Befehlssatz und über die Möglichkeit, auf verschiedene Speicherbereiche zuzugreifen, um Werte zu speichern und zu laden. Da es sich um eine abstrakte Maschine handelt, wird über die Spezifikation der JVM von Oracle keine direkte Implementierung beschrieben. Es werden lediglich die Richtlinien vorgegeben um eine funktionsfähige JVM zu implementieren. Eine der bekanntesten Implementierungen ist die *HotSpot JVM* von Oracle. Diese dient auch als Referenz-Implementierung.

Mit Hilfe der JVM können Programme, welche in Java oder einer anderen JVM-Sprache geschrieben wurden, auf beliebigen Plattformen verwendet werden, zumindest sofern eine JVM für die jeweilige Plattform bereitgestellt wird. Die JVM weiß nichts von den erzeugten Java-Klassen, welche ein Programm bilden. Mit Hilfe des Compilers werden die

Java-Klassen mit der Endung *.java* in *.class*-Dateien kompiliert. Diese Dateien bestehen aus JVM-Anweisungen, beziehungsweise JVM-Bytecode und zusätzlichen Informationen, welche später für die korrekte Ausführung der Programme benötigt werden. Der Aufbau dieser binären Dateien ist für Menschen nicht lesbar. Mit Hilfe des Tools *javap* können die Class-Dateien jedoch in eine lesbare Form gebracht werden. Zusätzlich erzeugt dieses Tool noch Kommentare zu den einzelnen JVM-Bytecodes. Dadurch wird die Lesbarkeit der Class-Dateien noch weiter erhöht. Im Anschluss werden wir den internen Aufbau der Lambda-Ausdrücke innerhalb der JVM und den Struktur der Class-Dateien näher betrachten.[LYBB15]

5.1.1 JVM-Datenbereiche

Während der Laufzeit legt die JVM mehrere Speicherbereiche für die Ausführung von Programmen fest. Da einige davon für die Analyse des Aufbaus der Lambda-Ausdrücke relevant sind, werden diese kurz vorgestellt. Hier werden ledliglich die relevanten Datenbereiche präsentiert.

- Die JVM verwendet, wie beispielsweise C, ebenfalls einen Stack. Innerhalb dieses Stacks werden zum Beispiel lokale Variablen und Ergebnisse von Berechnungen zwischengespeichert. Er wird aber ebenso für den Aufruf von Methoden verwendet, da sich die Parameter der Methoden auf dem Stack befinden. Ebenso wird der Rückgabewert der Methoden auf dem Stack gespeichert. Jeder JVM Thread verwendet einen eigenen, privaten Stack.[LYBB15]

- Der so genannten *Heap* wird von allen JVM Threads geteilt. Hier befindet sich der Speicher für das Anlegen von Klassen-Instanzen und für die Erzeugung von Arrays. Im Heap findet auch der Garbage-Collector seine Anwendung. Dabei wird der Speicher für Daten, die nicht mehr benötigt werden, nach einer gewissen Zeit vom Garbage-Collector freigegeben. Dadurch fällt im Vergleich zu C++ die manuelle Speicherverwaltung weg.[LYBB15]

- Im *Constant-Pool*, welcher jeweils zu einer Klasse beziehungsweise einem Interface gehört, werden verschiedene Arten von konstanten Daten abgespeichert. So befinden sich beispielsweise Namen von Klassen, aber auch Namen von Methoden in diesem Speicherbereich.[LYBB15]

5.1.2 JVM-Befehlssatz

Jeder Befehl, auch als *opcode* bezeichnet, wird über genau ein Byte dargestellt. Wie bei Assembler können auf den jeweiligen Befehl noch Operanden folgen, auf welche die gewünschte Operation angewandt werden soll. Über die Befehle wird auch gleichzeitig Typ-Information gegeben. So wird beispielsweise das Laden eines *int* über *iload* und das Laden eines *float*-Wertes über *fload* durchgeführt.[LYBB15]

Es werden Befehle für das Speichern von Daten auf den Stack und das Laden vom Stack bereitgestellt. Des Weiteren gibt es arithmetische Operationen, Befehle für Typ-Umwandlungen, Erstellung von Objekten und Arrays, spezielle Befehle für den Stack (beispielsweise *dup* um das oberste Element des Stacks zu duplizieren) und Befehle, welche Kontrollstrukturen repräsentieren.[LYBB15]

Eine für die Analyse von Lambda-Ausdrücken wichtige Art von Befehlen, die hier noch nicht erwähnt wurde, sind jene Befehle für die Ausführung von Methoden. Die folgenden Befehle für die Ausführung von Methoden werden von der JVM bereitgestellt.[LYBB15]

- *invokevirtual* wird verwendet, um die Ausführung von Methoden eines Objekts durchzuführen.
- *invokeinterface* wird für die Ausführung von Interface-Methoden eingesetzt.
- Über *invokespecial* wird die Ausführung von speziellen Methoden eines Objekts durchgeführt. Dazu zählt der Aufruf eines Konstruktors, einer privaten Methode oder einer Methode eines Obertyps der jeweiligen Klassen.
- *invokestatic* wird für den Aufruf von statischen Methoden verwendet.
- *invokedynamic* wurde mit Veröffentlichung von Java 7 zur JVM hinzugefügt. Dabei wird die tatsächlich ausgeführte Methode erst zur Laufzeit bestimmt. Diese Art der Methodenausführung wurde vor allem für dynamisch typisierte Sprachen zur JVM hinzugefügt. Sie spielt eine entscheidende Rolle im Bezug auf Lambda-Ausdrücke innerhalb von Java. Daher wird dieser Art des Methodenaufrufs noch ein eigenes Unterkapitel gewidmet.

Wie man anhand der ersten vier Aufrufmethoden, die bereits vor Java 7 vorhanden waren, sehen kann, wurden die Befehle der JVM für den Aufruf von Methoden sehr nah an jene der unterschiedlichen Methoden-Aufrufe in Java angelehnt.[Ros09]

5.2 invokedynamic

Bevor wir nun auf den Aufbau der Lambda-Ausdrücke innerhalb der JVM näher eingehen, werden zuerst die Konzepte von *invokedynamic* und der in diesem Zusammenhang ebenfalls neu hinzugekommenen *Method-Handles* betrachtet. In diesem Unterkapitel folgt zuerst eine kurze Einleitung zu *invokedynamic*. Anschließend wird veranschaulicht, warum dieser Befehl mit Java 7 zur JVM hinzugefügt wurde und wie er funktioniert.

Invokedynamic wurde über die JSR-292 im Zusammenhang mit der Plattform OpenJDK spezifiziert und über das Paper mit dem Titel *Bytecodes meet Combinators: invokedynamic on the JVM* von John R. Rose im Jahr 2009 ausführlicher vorgestellt.[Ros09]

Die JVM wurde in den vergangenen Jahren für die Implementierung neuer Sprachen zunehmend interessanter. Grund dafür sind einige Stärken, durch die sich die JVM

auszeichnet. Dazu zählen unter anderem die Optimierung durch den Compiler, die bereits erwähnte Garbage-Collection und die gute Skalierbarkeit im Bezug auf parallele Ausführung. Aber auch im Bezug auf die Sicherheit hat sich die Verwendung der JVM für neue Programmiersprachen hervorgehoben. Dennoch gab es bei der Entwicklung von Sprachen für die JVM einige Probleme.[Ros09]

Eines dieser Probleme steht im Zusammenhang mit dem Aufruf von Methoden innerhalb der JVM. Wie wir oben bereits gesehen haben, wurden die Möglichkeiten für Methodenaufrufe in der JVM sehr stark an Java angelehnt. Daher sind auch die Methodenaufrufe innerhalb der JVM statisch typisiert. Das bedeutet, dass sich innerhalb der Class-Dateien die Befehle und Methodenaufrufe auf exakte Typen beziehen. Daher mussten auch die Methodenaufrufe von anderen Sprachen mehr oder weniger an jene von Java angepasst werden, sofern man vor hatte, eine neue Sprache für die JVM zu implementieren. Ebenso wurden bereits bestehende Sprachen für den Einsatz auf der JVM adjustiert. So wurde beispielsweise Python unter dem Namen *Jython*, nachdem die Sprache in C implementiert wurde, auch für die JVM implementiert. Auf die Frage, warum Java und die JVM für diese Implementierung verwendet wurden, wird unter anderem mit der guten Portabilität und der hohen Robustheit der Sprache argumentiert[Hug97]. Auch die Sprache Ruby wurde als *JRuby* für die JVM-Plattform herausgebracht.[jru]

In [Ros09] werden Eigenschaften von Methoden aufgezählt. Über diese werden anschließend die Probleme, die es beim Methodenaufruf in anderen Sprachen gibt, definiert und über Beispiele spezifischer Sprachen verdeutlicht. Im Bezug auf Methoden wird zwischen Definition und Verwendung unterschieden. Da die Methodenaufrufe vor Java 7 keine dynamische Typisierung zugelassen haben, mussten dynamisch typisierte Sprachen eine zusätzliche Schicht zwischen Methodenaufruf innerhalb der JVM und tatsächlicher Ausführung der Methode in der Sprach-Laufzeit aufbauen, um die dynamischen Aspekte der Methodenaufrufe zu simulieren. Dazu wurden so genannte Methoden-Simulatoren verwendet, welche im nächsten Unterkapitel näher vorgestellt werden. So werden unter anderem Klassen in Smalltalk, JavaScript Scopes und optionale Argumente in Ruby als Beispiele genannt, wo durch die JVM nicht genügend Flexibilität entgegen gebracht wird. Des Weiteren können auch bei der Verlinkung zwischen einer Referenz und der auszuführenden Methode Situationen eintreten, die nicht ohne zusätzliche Maßnahmen innerhalb der JVM realisierbar sind, beziehungsweise vor Einführung von *invokedynamic* nicht ohne zusätzliche Indirektion realisierbar waren.

5.2.1 Methoden-Simulatoren

Vor der Einführung von *invokedynamic* wurde, um die oben aufgeführten Probleme beim Aufruf von Methoden zu umgehen, eine Reifikation von Methoden durchgeführt. Als Reifikation wird dabei die Umsetzung einer abstrakten Sache zu einer konkreten Sache bezeichnet. Die Reifikation von Methoden wurde dabei über die Simulation einer Methode über eine API umgesetzt. Dabei wird die auszuführende Methode über eine Simulationsmethode m innerhalb des dafür erzeugten Objektes s aufgerufen. Ebenso kann auch an jener Stelle, an welcher die Methode aufgerufen wird, eine solche Simulation

verwendet werden. Es kann also auch hier ein zusätzliches Objekt, welches dann für den Methodenaufruf zuständig ist, eingesetzt werden. Die Stelle, von der aus eine Methode aufgerufen wird, wird als *Callsite* bezeichnet. Dadurch, dass der Methodenaufruf über ein speziell dafür angelegtes Objekt durchgeführt wird, kann das Verhalten des Methodenaufrufs beliebig verändert werden. Dieser Mechanismus wurde beispielsweise von JRuby für die Umgehung der geringen Flexibilität der Methodenaufrufe innerhalb der JVM verwendet.[Ros09]

Der Nachteil bei dieser Variante ist der zusätzliche Mehraufwand, welcher bei der Ausführung der Methoden anfällt. Zusätzlich gewinnt die Ausführung der Methoden auch an Komplexität. Im Vergleich zum direkten Methodenaufruf innerhalb der JVM, durch eine der vier *invoke*-Befehle, fällt es der JVM hier wesentlich schwerer, den Code für die Ausführung der Methode zu optimieren.[Ros09]

5.2.2 Funktionsweise von invokedynamic

Der Befehl *invokedynamic* wird bei der ersten Ausführung der dynamischen Callsite mit der auszuführenden Methode verbunden. Eine dynamische Callsite entspricht einer Instanz von *invokedynamic*, beziehungsweise einer Reifikation von *invokedynamic*. So wird beispielsweise in Java der Befehl *invokedynamic* über *java.lang.invoke.CallSite* reifiziert. Ohne die in *java.lang.invoke* definierten Klassen könnte *invokedynamic* nicht funktionieren. Betrachtet man außerdem die früheren Möglichkeiten der JVM um Methoden aufzurufen, so kann man Folgendes beobachten. Methodenaufrufe über beispielsweise *invokestatic* können direkt über eine statische Methode in Java innerhalb der Sprache repräsentiert werden. Bei *invokedynamic* ist dies nicht direkt möglich.

Die Klasse, in welcher die dynamische Callsite vorkommt, registriert bei der JVM eine so genannte Bootstrap-Methode. Diese Methode ist dann Teil des Constant-Pools der Class-Datei. Vor der ersten Ausführung der Methode wird von der JVM die festgelegte Bootstrap-Methode aufgerufen. Über diese wird, über die Laufzeit der jeweiligen Sprache, die dynamische Callsite als Objekt *c* reifiziert. Das ist auch der Zeitpunkt, wo die sprachspezifische Logik, welche für den Aufruf der Methode benötigt werden kann, ausgeführt wird. Über die Bootstrap-Methode wird das Callsite-Objekt an die JVM zurückgegeben. Daher wird diese Methode auch als Callsite-Fabrik bezeichnet. Das Callsite-Objekt beinhaltet einen Method-Handle. Über diesen wird die aufzurufende Methode festgelegt. Die Laufzeit der Sprache bestimmt über das Callsite-Objekt. So kann die Zielmethode über das Callsite-Objekt mittels des Method-Handles neu gesetzt werden. Des Weiteren kann die dynamische Callsite vom Objekt getrennt werden. Danach muss diese jedoch, vor einer erneuten Ausführung, über die Bootstrap-Methode neu verlinkt werden. Nachdem eine Callsite verlinkt wurde, kann die assoziierte Methode direkt über den Method-Handle ausgeführt werden. Das bedeutet, sobald *invokedynamic* mit einer CallSite reifiziert wurde, werden alle Aufrufe von *invokedynamic* über die CallSite direkt an die jeweilige Methode des dazugehörigen Method-Handles weitergereicht.[Ros09]

Im Zusammenhang mit *invokedynamic* wurde auch ein neuer Mechanismus hinzuge-

fügt, über den Methoden gesucht und ausgeführt werden können. Somit wurden Zeiger auf Funktionen zur JVM hinzugefügt[Ros09]. Dieser Mechanismus wird im nächsten Unterkapitel näher vorgestellt.

5.3 Method-Handles

In diesem Unterkapitel werden Method-Handles genauer vorgestellt. Diese wurden im Zusammenhang mit *invokedynamic* zu Java 7 hinzugefügt. Method-Handles wurden deshalb hinzugefügt, weil im Bezug auf *invokedynamic* eine Datenstruktur benötigt wurde, mit deren Hilfe direkt eine beliebige Methode referenziert und dadurch auch ausgeführt werden kann. Dadurch kann dynamisch zur Laufzeit über *invokedynamic* eine Verbindung zwischen der Call-Site und der auszuführenden Methode erzeugt werden[Ros09]. Eine weitere Möglichkeit wäre *Method* von der *Core Reflection API* gewesen. Doch dadurch hätte die Performance gelitten[ORPS09]. Bei der Reflection API hätte bei jedem Aufruf der Methode überprüft werden müssen, ob der Aufrufer auch die benötigen Rechte dafür besitzt. Die Rechte werden dabei immer relativ zur aufrufenden Klasse überprüft. Im Gegensatz dazu werden bei Method-Handles nur bei deren Erzeugung die Aufrufberechtigungen abgefragt. Diese werden immer an jene Klasse gebunden, welche den Method-Handle erzeugt. Daher muss beim Weitergeben dieser zu anderen Klassen bedacht werden, dass dadurch private Methoden der Ersteller-Klasse aufrufbar sind. Der einzig relevante Aspekt für die dynamische Call-Site im Bezug auf den Aufruf von Methoden, welche dieser dynamisch zur Laufzeit zugeteilt werden, ist der schnellstmögliche Aufruf dieser[Ros09].

Die Umsetzung von Method-Handles auf Sprachebene von Java wird im Paket *java.lang.invoke* in der Klasse *MethodHandle* festgelegt. Die dazugehörige Klasse *MethodHandles* bietet statische Methoden zur Verarbeitung und Erzeugung von Method-Handles an[Orab]. Um die Verwendung der Method-Handles innerhalb von Java besser nachvollziehen zu können, wird die Funktionsweise dieser über das folgende Beispiel vorgestellt. In diesem Beispiel soll die Methode *replace* über ein Method-Handle aufgerufen werden.

```
//Beispiel: Method-Handles in Java
MethodHandles.Lookup lookup = MethodHandles.lookup();
MethodType mt = MethodType.methodType(String.class,
                                      char.class,
                                      char.class);
MethodHandle mh = lookup.findVirtual(String.class,
                                     "replace",
                                     mt);
String s = (String)mh.invokeExact("aaa",'a','b');
```

Über *MethodHandles.lookup* wird ein neues Objekt vom Typ *Lookup* zurückgeliefert. Bei der Erzeugung wird die Klasse, durch die der Lookup erzeugt wurde, gespeichert, um später

bei der Erzeugung von Method-Handles die Zugriffsberechtigungen zu kontrollieren[Orab]. Anschließend können über das Lookup-Objekt neue Method-Handles erzeugt werden. Als nächstes wird in diesem Beispiel der Methodentyp deklariert. Dabei wird zuerst der Rückgabewert als *String* festgelegt. Die beiden Argumente sind vom Typ *char*. Der Methoden-Typ entspricht also exakt der Signatur der gesuchten Methode. Nachdem der Methoden-Typ festgelegt wurde, kann nun mit Hilfe des Lookup-Objekts und der dazugehörigen Methode *findVirtual* nach der Methode *replace* gesucht werden. Die Argumente für *findVirtual* sind dabei wie folgt festgelegt. Über das erste Argument wird die Klasse beziehungsweise das Interface angegeben. Über diese Klasse kann die Methode aufgerufen werden. Anschließend wird der Name der Methode angegeben. Danach folgt der Methoden-Typ. Die Methode *replace* kann nun über den Method-Handle mit *invokeExact* aufgerufen werden. Bei *invokeExact*, neben *invoke* die einzigen beiden Methoden mit dieser Eigenschaft[LYBB15], handelt es sich um eine signatur-polymorphe Methode. Das bedeutet, dass die Argumente und der Rückgabewert vom Typ *Object* sind. Daher muss das Resultat von *replace* explizit in einen String umgewandelt werden. Der String *aaa* wird durch Ersetzung mittels *replace* zum String *bbb*.[Orab]

Im Bezug auf den Bytecode sieht man hier einen Unterschied zwischen dem Einsatz von Method-Handles und dem direkten Aufruf der Methode. Bei jenem Beispiel mit Method-Handles wird im Bytecode mit *invokevirtual* die Methode *invokeExact* ausgeführt. Würde man im Vergleich dazu die Methode *replace* direkt aufrufen, so würde im Bytecode über *invokevirtual* die Methode *replace* aufgerufen werden. Nachfolgend werden die wichtigsten Eigenschaften von Method-Handles vorgestellt. In [Ros09] wurden die folgenden Prinzipien für Method-Handles festgelegt.

- Über Method-Handles soll der direkte Aufruf von Methoden der JVM ermöglicht werden. Dazu zählt auch der Zugriff auf private Methoden.
- Method-Handles sollen funktional durch Komposition verbunden werden können. So können mit Hilfe der statischen Methoden der Klasse *MethodHandles.java* beispielsweise Argumente hinzugefügt werden. Des Weiteren können Argumente für den Methodenaufruf gefiltert oder verworfen werden. Es ist auch möglich, vorbereitete Argumente in einem Method-Handle einzufügen und sich danach einen neuen Method-Handle zurückgeben zu lassen, wobei dieser dann den eingefügten Wert beinhaltet. So würde man beim vorherigen Beispiel mit Hilfe von *MethodHandles.insertArguments(mh,0,"test")* einen neuen Method-Handle zurückbekommen. Beim anschließenden Aufruf der Methode durch diesen, wird als erstes Argument der Wert *test* vom Typ *String* übergeben. Dadurch benötigt man beim Ausführen der Methode nur noch zwei Argumente.
- Method-Handles sollen polymorph in Hinsicht auf alle JVM Signaturen sein. Dazu werden die beiden Methoden zum Aufruf der Methode *invoke* und *invokeExact* mit *PolymorphicSignature* annotiert. Die Argumente sind vom Typ *Object* und ebenso ist der Rückgabewert vom Typ *Object*. Es können dadurch beliebige Typen als Eingabe- und Ausgabeparameter verwendet werden.[Orab]

- Der einheitliche Zugriff auf JVM-Befehle soll ermöglicht werden, dazu gehören vor allem die *invoke*-Befehle, durch welche Methoden aufgerufen werden können.

- Die Argumente sollen über Casting der Typen beziehungsweise Boxing umgewandelt werden können. Dies wird durch die Methode *invoke*, im Gegensatz zu *invokeExact*, verwirklicht. Dadurch werden Argumente beim Aufruf der Methode automatisch in die passenden Typen umgewandelt sofern dies möglich ist. Genauso verhält es sich beim Rückgabewert.

- Partielle Anwendung soll ermöglicht werden. Dies bezieht sich auf den Aufruf von Methoden. Durch die partielle Anwendung auf eine Methode wird es ermöglicht, diese mit weniger Argumenten auszuführen.

5.4 Lambda-Ausdrücke in der JVM

In diesem Abschnitt werden wir uns mit der Umsetzung der Lambda-Ausdrücke innerhalb der JVM beschäftigen. Um besser nachvollziehen zu können, warum sich die Entwickler für den Einsatz von *invokedynamic* im Zusammenhang mit der Darstellung von Lambda-Ausdrücken innerhalb der JVM entschieden haben, werden wir uns zuerst mögliche Alternativen ansehen. Eine Möglichkeit wäre gewesen, die Lambda-Ausdrücke intern als innere Klassen umzusetzen, ebenso wie bei anonymen inneren Klassen. Dadurch wären die Lambda-Ausdrücke quasi nur syntaktischer Zucker für anonyme innere Klassen gewesen. Eine andere Umsetzung wäre über die Verwendung von Method-Handles realisierbar geworden. Um die letztendlich gewählte Methode mit *invokedynamic* und deren Vorteile besser zu verstehen, wird zuerst der Aufbau anonymer innerer Klassen als Bytecode der JVM betrachtet. Dazu wird das mit Java mitgelieferte Tool *javap* verwendet. Dabei handelt es sich um einen Disassembler für *.class*-Dateien[Orag]. Dadurch werden diese in eine für Menschen lesbare Form gebracht und wir können den internen Aufbau der Klassen in der JVM näher analysieren.

Im April 2012 wurde von Brian Goetz, der mit der Leitung der Umsetzung von Lambda-Ausdrücken beauftragt wurde, ein Paper mit dem Titel *Translation of Lambda Expressions* veröffentlicht[Goe12b]. In diesem wird ausführlich über die Übersetzung der Lambda-Ausdrücke gesprochen. Dabei wird auch darauf eingegangen, dass es verschiedene Möglichkeiten im Bezug auf deren Umsetzung in der JVM gibt. Jede dieser hat gewisse Vor- und Nachteile. Die Entscheidung fiel letztlich über die durch die Expertengruppe des JSR-335 gesetzten Ziele bezüglich der Bytecode-Repräsentation von Lambda-Ausdrücken. Zum einen soll die Flexibilität für eine spätere Optimierung der Umsetzung der Lamda-Ausdrücke gegeben sein. Dies kann nur dadurch erfüllt werden, dass man sich nicht auf eine spezielle Umsetzung als Bytecode festlegt. Gleichzeitig soll aber auch die Repräsentation innerhalb der Class-Dateien stabil bleiben. Sobald man sich also für eine Bytecode-Repräsentation entschieden hat, muss diese auch in nachfolgenden Versionen eingehalten werden[Goe12a]. Grund dafür ist die Rückwärtskompatibilität von Java.[Goe12b]

5.4.1 Bytecode anonymer innerer Klassen

Hier betrachten wir zunächst die Umsetzung von anonymen inneren Klassen in der JVM. Für den erzeugten Bytecode wird der folgende Code verwendet. Dabei wird in der Klasse eine Funktion *foo* definiert. Innerhalb dieser wird ausführbarer Code in Form der Variable *runnable* festgelegt. Es soll der String *test* über *println* ausgegeben werden. In diesem Fall wird für die Deklaration von *runnable* kein Lambda-Ausdrück, sondern eine anonyme innere Klasse verwendet. Mit *runnable.run* wird der in *run* definierte Code anschließend ausgeführt.

```
public class B {
    public void foo() {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("test");
            }
        };

        runnable.run();
    }
}
```

Die Klasse *B.java* wird über den Compiler in die beiden Class-Dateien umgewandelt. Diese heißen *B.class* und *B$1.class*. Die anonyme innere Klasse wird also als eigene Class-Datei angelegt. Im Folgenden sehen wir die Ausgabe durch die Anwendung des Disassemblers *javap -c -v B.class*. Dadurch wird der Code in JVM-Bytecode dargestellt. Über die Option *v* werden zusätzliche Informationen angezeigt, wie beispielsweise der Speicherbereich der Konstanten. Dieser wird nachfolgend als *Constant Pool* bezeichnet. Auf Grund der Länge der Ausgabe werden im Folgenden nur die relevanten Teile der Ausgabe von *javap* dargestellt. Zur besseren Lesbarkeit werden die Referenzen zum Constant-Pool aufgelöst.

```
public B();
  flags: ACC_PUBLIC
  Code:
    0: aload_0
    1: invokespecial Method java/lang/Object."<init>":()V
    4: return

public void foo();
  flags: ACC_PUBLIC
  Code:
```

```
 0: new           class B$1
 3: dup
 4: aload_0
 5: invokespecial Method B$1."<init>":(LB;)V
 8: astore_1
 9: aload_1
10: invokeinterface InterfaceMethod Runnable.run:()V
15: return
```

Hier sieht man den Bytecode der beiden Methoden, die sich innerhalb der Klasse befinden. Bei der Methode *public B* handelt es sich um den Konstruktor der Klasse. Dieser wurde von *Object* geerbt. Mit *aload_0* wird *this* auf den Stack gepusht[LYBB15]. Über *invokespecial* wird der Konstruktor der Klasse aufgerufen. Dieser verwendet die vorher auf den Stack gepushte Referenz auf *this*.

Der für uns interessante Teil befindet sich im Bytecode der Methode *foo*. Hier sieht man bei 0, dass ein neues Objekt für die anonyme innere Klasse *B$1* erzeugt wird. Normalerweise würden hier Nummern neben den Instruktionen stehen, wie beispielsweise #2 nach dem Befehl *new*. Diese beziehen sich auf Referenzen auf den Constant-Pool. Auf Grund der besseren Lesbarkeit wurden diese Referenzen in diesem Beispiel bereits aufgelöst. Über das Tool *javap* wird in Form von Kommentaren gezeigt, was sich im dazugehörigen Index des Constant-Pools befindet. Mit *invokespecial* wird der Konstruktor für die innere Klasse aufgerufen. Dies wird durch den Verweis auf die Konstante Method <init> dargestellt. Mit *astore_1* wird der oberste Wert des Stacks in die lokale Variable 1 gespeichert. Damit die Methode *runnable.run* ausgeführt werden kann, wird über *aload_1* die Instanz der inneren Klasse auf den Stack geladen und anschließend wird mit *invokeinterface* die Methode *run* des Interfaces *Runnable* ausgeführt. Für genauere Beschreibungen der Befehle der JVM sei hier auf die aktuelle JVM Spezifikation für Java 8 verwiesen[LYBB15]. Nachfolgend sieht man den Bytecode der inneren Klasse *B$1*.

```
public void run();
  Code:
    0: getstatic     Field System.out:PrintStream;
    3: ldc           String test
    5: invokevirtual Method PrintStream.println:(String;)V
    8: return
```

Hier wird *System.out* über *getstatic* angefordert. Anschließend wird über *ldc* der String *test* auf den Stack geladen und anschließend wird mit *invokevirtual* die Methode *println*, mit dem String als Argument und *void* als Rückgabewert, ausgeführt.

Bei dieser Art der Umsetzung gibt es die folgenden Nachteile. Zum einen würde das Ziel für die Flexibilität[Goe12b] im Zusammenhang mit der fixen Darstellung der Lambda-Ausdrücke nicht erreicht werden. Hätte sich das Team rund um Brian Goetz für diese

Art der Umsetzung von Lambda-Ausdrücken entschieden, so würde es nachfolgend keine Möglichkeit geben, diese JVM-Bytecode-Repräsentation ändern zu können[Goe12a]. Lambda-Ausdrücke wären für immer als anonyme innere Klassen dargestellt worden. Zusätzlich dazu gibt es generelle Nachteile für die Verwendung von inneren Klassen. Die innere Klasse wird als eigene Class-Datei angelegt. Demnach würde, sofern dieser Ansatz für die Darstellung als JVM-Bytecode gewählt worden wäre, für jeden Lambda-Ausdruck eine eigene innere Klasse angelegt werden. Folglich würde dadurch auch jedes Mal eine neue Class-Datei im Dateisystem erzeugt werden. Und dann wären auch die *jar*-Dateien, bei denen es sich um Java-Archive handelt, wobei die darin beinhalteten Java-Dateien für die Ausführung von Code verwendet werden können, auf Grund der zusätzlichen Class-Dateien größer. Dadurch ergeben sich wiederum Nachteile für die Performance. Grund dafür ist, dass für die inneren Klassen jedes mal ein neues Objekt angelegt wird und anschließend die Klasse durch den Class-Loader geladen werden muss[LB98]. Ein weiterer Nachteil ist die im Zusammenhang mit inneren Klassen komplizierte Namensgebung der Class-Dateien[Goe12a].

Wenn für jeden Lambda-Ausdruck eine eigene Klasse angelegt werden muss, so kann es bei häufiger Verwendung ein und desselben funktionalen Interfaces, zu einer so genannten *Type profile pollution* kommen[Goe12a]. So wird bei der JVM-Implementierung *HotSpot* ein Typ-Profil angelegt. Solche Arten von Profilen werden in der JVM für Optimierungen des Bytecodes eingesetzt[Ros13b]. Dabei wird das Verhalten eines Bytecode-Befehls im Profil zusammengefasst. Wenn eine gewisse Methode immer mit denselben Typen durchgeführt wurde, so ist die Wahrscheinlichkeit sehr hoch, dass auch bei zukünftigen Aufrufen der gleiche Typ verwendet wird. Dadurch kann der Aufruf einer Methode über die Profilierung durch die JVM optimiert werden, beispielsweise durch Inlining. Wenn ein solcher Methodenaufruf jedoch mit unterschiedlichen Typen aufgerufen wird, so wird diese Optimierung nicht möglich sein und der Compiler muss bei jedem Aufruf die Typen überprüfen. Dies ist als Profil-Verschmutzung bekannt und führt zu einem Nachlassen der Performance. Wenn wir nun beispielsweise hunderte *Runnable*-Implementierungen verwenden, so wird es zu einer *Verschmutzung* des Typ-Profils kommen und die Optimierungen durch die JVM können nicht mehr durchgeführt werden.[Ros13a]

5.4.2 Lambda-Ausdrücke über Method-Handles

Eine weitere Möglichkeit wäre es gewesen, anstelle von inneren Klassen, die in Java 7 neu hinzugekommenen Method-Handles zu verwenden. Die Vor- und Nachteile dieses Ansatzes wurden von Brian Goetz in *Lambda: A Peek Under The Hood* auf der JAX 2012 in London vorgestellt[Goe12a]. Dabei wurde die Umsetzung von Lambda-Ausdrücken in der JVM mit Hilfe von Method-Handles als offensichtliche Möglichkeit betrachtet, da es sich bei Lambda-Ausdrücken auf der Sprach-Ebene um ein Objekt mit einer Methode handelt und Method-Handles ebenfalls genau dieses Konzept umsetzen. Daher wäre es möglich gewesen, die Lambda-Ausdrücke zuerst in Methoden umzuwandeln und anschließend mit Hilfe von Method-Handles den Lambda-Ausdruck über Bytecode darzustellen. Das Problem, das sich dabei ergibt, soll mit folgendem Beispiel veranschaulicht werden.

In diesem Beispiel gehen wir davon aus, dass Lambda-Ausdrücke mit Hilfe von Method-Handles innerhalb der JVM umgesetzt werden. Wir übergeben der Methode *list.remove* ein Prädikat in Form eines Lambda-Ausdrucks. Dieser Lambda-Ausdruck wird anschließend für die Darstellung als JVM-Bytecode in eine statische Methode umgewandelt. Dessen Argumente sind jene des Lambda-Ausdrucks und zusätzlich die Variablen, die vom umliegenden Geltungsbereich innerhalb des Funktionskörpers des Lambda-Ausdrucks verwendet werden. Es handelt sich hierbei um eine Mischung aus Java- und Pseudo-Code, da sich *LDC[lambda$1]* nicht in Java-Code darstellen lässt.[Goe12a]

```
list.removeIf(p -> p.age < minAge);

//wird folgendermaßen über die JVM umgesetzt
private static boolean lambda$1(int minAge, Person p) {
    return p.age < minAge;
}

MethodHandle mh = LDC[lambda$1];
mh = MethodHandles.insertArguments(mh, 0, minAge);
list.removeIf(mh);
```

Hier sieht man, wie aus dem in *list.removeIf* als Prädikat verwendeten Lambda-Ausdruck zuerst eine statische Methode erzeugt wird. Anschließend wird ein MethodHandle erzeugt. Dabei werden die von außen benötigten Variablen als Argumente über *insertArguments* zum MethodHandle hinzugefügt. Soweit sieht diese Umsetzung von Lambda-Ausdrücken zu Method-Handles problemlos aus. Es ergeben sich dabei aber die folgenden Nachteile, die letztlich gegen die direkte Verwendung von Method-Handles für die Umsetzung von Lambda-Ausdrücken in der JVM geführt haben. Die Signatur von *removeIf* ändert sich auf Grund der Method-Handles folgendermaßen[Goe12a]:

```
//Standard-Signatur removeIf
boolean removeIf(Predicate predicate)
//Signatur nach Umwandlung zu MethodHandle
boolean removeIf(MethodHandle predicate)
```

Durch die Änderung der Signatur wäre es nicht mehr möglich eine Methode mit zwei unterschiedlichen Lambda-Ausdrücken überladen zu können. Zwei unterschiedliche Lambda-Ausdrücke würden nach der Umwandlung als Objekt vom Typ *MethodHandle* vorkommen. Überladung wäre daher nicht möglich, da es bei der Umwandlung zu einem Method-Handle, ähnlich wie bei *Generics*[Niñ07], zu einer Typlöschung kommt. Ebenso hätte man bei dieser Variante den Bytecode für die Umsetzung von Lambda-Ausdrücken fix festgelegt und hätte im Gegensatz zur Umsetzung mit *invokedynamic* auf die gewünschte Flexibilität[Goe12b] verzichtet.[Goe12a]

5.4.3 Lambda-Ausdrücke über invokedynamic

In diesem Unterkapitel wird die Umsetzung der Lambda-Ausdrücke innerhalb der JVM mit dem neuen Methodenaufruf-Mechanismus *invokedynamic* vorgestellt. An Hand mehrerer Beispiele soll die Funktionsweise und der Aufbau innerhalb der JVM analysiert werden.

Das nachfolgende Beispiel zeigt den erzeugten JVM-Bytecode beim Einsatz von Lambda-Ausdrücken in Java 8. Dabei soll über einen Consumer vom Typ *String* ein String an *System.out.println* weitergegeben werden. Der folgende Programmcode wird mit *javap* analysiert. Zuerst betrachten wir den Aufbau und die Umsetzung eines zustandlosen Lambda-Ausdrucks. Dabei werden keine Variablen von der umschließenden Klasse verwendet. Hier könnte man den Lambda-Ausdruck durch eine Methoden-Referenz ersetzen. Da wir uns hier jedoch die Umsetzung der Lambda-Ausdrücke anschauen wollen, werden wir zuerst den Lambda-Ausdruck analysieren. Danach können wir die Unterschiede im Vergleich zur Methoden-Referenz betrachten. Anschließend wird erklärt wie die Umsetzung beim Zugriff des Lambda-Ausdruckes auf Werte nach außen funktioniert.

```
public class A {
    public void foo() {
        Consumer<String> consumer = s -> System.out.println(s);
        consumer.accept("test");
    }
}
```

In der Klasse *A* befindet sich die Methode *foo*. Über diese wird der Lambda-Ausdrück dem funktionalen Interface *Consumer* zugewiesen. Anschließend wird dieser Lambda-Ausdruck über *consumer.accept* ausgeführt und gibt den String *test* aus.

Lambda-Umwandlung in eine Methode

Der erste Schritt, der bei der Übersetzung der Lambda-Ausdrücke durch den Compiler durchgeführt wird, ist Folgender. Der Lambda-Ausdruck wird durch den Compiler auf eine herkömmliche Methode reduziert. Dazu wird der Funktionskörper des Lambda-Ausdrucks als Implementierung für die Methode verwendet. Die Parameter für die Methode werden dabei ebenfalls übernommen. Die erzeugte Methode kann jedoch, auf Grund von Zugriffen auf Variablen, welche sich außerhalb des Lambdas befinden, zusätzliche Parameter beinhalten. Bei der Umwandlung in eine Methode gibt es derzeit zwei Strategien. Bei zustandslosen Lambda-Ausdrücken wird eine statische, private Methode erzeugt, deren Signatur jener des Lambda-Ausdrucks entspricht. In unserem Beispiel würde also die folgende Methode in der Klasse *A* erzeugt werden.[Goe12b]

```
private static void lambda$foo$0(String s) {
    System.out.println(s);
}
```

Die statische Methode nimmt einen String entgegen und gibt diesen über *println* aus, genauso wie der Lambda-Ausdruck, beziehungsweise das damit implementierte funktionale Interface *Consumer*. Betrachten wir nun den JVM-Bytecode, der über *javap -v -c A.class* in eine lesbare Form gebracht wurde. Um zusätzlich dazu die durch den Compiler für den Lambda-Ausdruck erzeugte Methode zu sehen, verwenden wir noch zusätzlich das Argument *-private*. Nicht relevante Informationen werden aus Gründen der Lesbarkeit nicht angegeben. Dazu gehört der relativ lange Pool der Konstanten, der am Anfang der Class-Datei, nach der BootstrapMethode, stehen würde. Sofern für die Erklärung des Bytecodes Informationen aus dem Constant-Pool benötigt werden, werden Teile davon während der Analyse erwähnt.

```
BootstrapMethods:
    0: #29 invokestatic
      java/lang/invoke/LambdaMetafactory.metafactory:(
        Ljava/lang/invoke/MethodHandles$Lookup;
        Ljava/lang/String;Ljava/lang/invoke/MethodType;
        Ljava/lang/invoke/MethodType;
        Ljava/lang/invoke/MethodHandle;
        Ljava/lang/invoke/MethodType;
        ) Ljava/lang/invoke/CallSite;
      Method arguments:
        #30 (Ljava/lang/Object;)V
        #31 invokestatic A.lambda$foo$0:(Ljava/lang/String;)V
        #32 (Ljava/lang/String;)V

//Hier wäre der Constant-Pool

public A();
    flags: ACC_PUBLIC
    Code:
      0: aload_0
      // Method java/lang/Object."<init>":()V
      1: invokespecial #1
      4: return

public void foo();
  flags: ACC_PUBLIC
  Code:
    // InvokeDynamic #0:accept:()Ljava/util/function/Consumer;
    0: invokedynamic #2,  0
    5: astore_1
    6: aload_1
    7: ldc             String test
```

```
    9: invokeinterface Consumer.accept:(Ljava/lang/Object;)V
    14: return

private static void lambda$foo$0(java.lang.String);
  flags: ACC_PRIVATE, ACC_STATIC, ACC_SYNTHETIC
  Code:
    // Field java/lang/System.out:Ljava/io/PrintStream;
    0: getstatic     #5
    3: aload_0
    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    4: invokevirtual #6
    7: return
```

Wie bei der Umsetzung der anonymen inneren Klassen in JVM-Bytecode, wird auch hier ein Konstruktor für die Klasse *A* erzeugt. Anschließend befindet sich die Definition der Methode *public void foo()*. Die Kommentare wurden aus Platzgründen vor die jeweilige Instruktion verschoben. Betrachten wir zunächst die synthetische Methode, die für unseren Lambda-Ausdruck erzeugt wurde. Aus dem Lambda-Ausdruck wurde eine private statische Methode erzeugt, die einen String entgegennimmt und als Rückgabewert *void* zurückgibt. Innerhalb dieser Methode soll, wie bereits erwähnt, der übergebene String über *println* ausgegeben werden. Diese Methode wird im späteren Verlauf, nämlich beim Aufruf von *invokedynamic* als Argument an die Bootstrap-Methode übergeben. Mit *getstatic* wird der *PrintStream* von *System.out* auf den Stack gepusht. Mit *aload_0* wird in statischen Methoden das erste Argument, also der übergebene String, auf den Stack gepusht[LYBB15]. Über *invokevirtual* wird anschließend *println* mit dem vorher gepushten String durchgeführt.

Als nächstes schauen wir uns die Methode *foo* an, in der Folgendes passiert. An jener Stelle im Code, an welcher der Lambda-Ausdruck vorkommt, wird eine dynamische Call-Site über *invokedynamic* erzeugt. Der Vorteil von *invokedynamic* ist, dass die Auswahl der Übersetzungs-Strategie nicht schon vom Compiler bestimmt wird. Stattdessen wird diese Entscheidung auf die Laufzeit des Programms hinausgezögert[Goe12b]. Dadurch legt man sich nicht hundert-prozentig auf die Umsetzung von Lambda-Ausdrücken innerhalb der JVM fest. Im Sinne des Bytecodes wird *invokedynamic* verwendet. Es wird lediglich ein Rezept für die Erstellung des Lambda-Ausdrucks an *invokedynamic* übergeben. Dahinter verbirgt sich jedoch eine dynamisch austauschbare Übersetzungsstrategie für Lambdas. Diese Strategie wird dem Laufzeitsystem von Java überlassen. Dies geschieht durch die Festlegung von Schnittstellen. Dahinter liegende Implementierung kann von Version zu Version ausgetauscht werden. Des Weiteren können auch mehrere Implementierungen für unterschiedliche Typen von Lambda-Ausdrücken bereitgestellt werden. Diese können dann auch während der Laufzeit ausgewählt werden. Dadurch werden die durch [Ros09] festgelegten Ziele, die bereits weiter oben erwähnt wurden, erreicht.

In unserem Beispiel wird die JVM-Instruktion *invokedynamic* mit den Argumenten #2

und 0 aufgerufen. Dabei bezieht sich #2 auf den zweiten Eintrag im *Constant-Pool*. Der von *javac* hinzugefügte Kommentar zeigt uns bereits, was sich in diesem Eintrag befindet. Die folgenden Informationen zur Spezifikation von *invokedynamic* sind der JVM-Spezifikation entnommen[LYBB15]. Der Befehl *invokedynamic* verweist auf einen Eintrag des *Constant-Pool*. Dort befinden sich die zur Laufzeit benötigten Konstanten. An jenen Index des *Constant-Pools*, auf den verwiesen wird, muss sich eine symbolische Referenz auf einen *Call Site Specifier* befinden. Diese Referenz wird von der JVM über die Struktur *CONSTANT_InvokeDynamic_info* bereitgestellt. Zum einen befindet sich darin eine Referenz auf einen Method-Handle. Dieser dient dem Befehl *invokedynamic* als Bootstrap-Methode. Um die Bootstrap-Methode festlegen zu können werden außerdem der auszuführende Methodenname und eine dazugehörige Signatur benötigt[LYBB15]. In unserem Beispiel sieht *InvokeDynamic* im Constant-Pool folgendermaßen aus. Wobei hier die zweite Referenz bereits aufgelöst wurde.

```
#2 = InvokeDynamic #0:accept:()Ljava/util/function/Consumer;
```

Der Verweis auf die Bootstrap-Methode wird über #0 durchgeführt. Danach befindet sich der Name der auszuführenden Methode *accept*. Die Signatur der Methode setzt sich im Zusammenhang mit der Erzeugung von Lambda-Ausdrücke wie folgt zusammen. In der Klammer befinden sich normalerweise die Parametertypen der Methode *accept*[LYBB15]. Jedoch wird die Methodensignatur hier zu einem anderen Zweck verwendet. Dies wird in der Dokumentation der LamdaMetafactory beschrieben[Orab]. Die Parametertypen beziehen sich in diesem Fall, auf die von außerhalb des Lambda-Ausdrucks erfassten Variablen. Da es sich in diesem Beispiel um einen zustandslosen Lambda-Ausdruck handelt, sind keine Parameter innerhalb der Klammern. Danach wird das zu implementierende Interface für die Bootstrap-Methode angegeben (*Consumer*).

Bootstrap-Methode

Hier soll die Bootstrap-Methode näher erläutert werden. Dazu betrachten wir zunächst jene Bootstrap-Methode, welche innerhalb unseres Beispiels verwendet wird. In diesem Zusammenhang wird die so genannte Lambda-Metafactory vorgestellt. Diese stellt die Schnittstelle auf Seiten der Java-Laufzeit für die Übersetzungsstrategie der Lambda-Ausdrücke dar.

```
BootstrapMethods:
    0: #29 invokestatic
      java/lang/invoke/LambdaMetafactory.metafactory:(
          Ljava/lang/invoke/MethodHandles$Lookup;
          Ljava/lang/String;
          Ljava/lang/invoke/MethodType;
          Ljava/lang/invoke/MethodType;
          Ljava/lang/invoke/MethodHandle;
```

```
        Ljava/lang/invoke/MethodType;
    )Ljava/lang/invoke/CallSite;
    Method arguments:
        #30 (Ljava/lang/Object;)V
        #31 invokestatic A.lambda$foo$0:(Ljava/lang/String;)V
        #32 (Ljava/lang/String;)V
```

Für die Umsetzung von Lambda-Ausdrücken unter Java 8 in der JVM wird die statische Methode *metafactory* in der Klasse *LambdaMetafactory* verwendet. Diese befindet sich im Paket *java.lang.invoke*. Da es sich um eine statische Methode handelt wird diese über den JVM-Befehl *invokestatic* ausgeführt. Über die Metafactory wird die Erzeugung der Objekte, die der Implementierung des jeweiligen funktionalen Interfaces entsprechen, vereinfacht. Die statische Methode *metafactory* liefert ein Objekt vom Typ *CallSite* zurück. Die folgenden sechs Argumente, wie man oben anhand der Methodensignatur sehen kann, werden für die Erstellung der CallSite benötigt.[Orab]

- Das erste Argument *caller* ist vom Typ *MethodHandles.Lookup*. Ein Lookup-Objekt wird, wie wir bereits im Rahmen dieser Arbeit im Abschnitt *Method-Handles* gesehen haben, dazu benötigt, um Method-Handles zu erstellen. Hier wird der Metafactory ein Lookup jener Klasse übergeben, über welche die dynamische Call-Site erzeugt wurde, also die Klasse, in der unser Lambda-Ausdruck definiert und dadurch der Aufruf von *invokedynamic* erzeugt wurde. Dabei werden im Bezug auf die Erstellung der Method-Handles die Berechtigungen der jeweiligen Klasse übergeben. Dieses Argument wird von der JVM bereitgestellt[Orab]
- Über das Argument *invokedName* wird der Name der aufzurufenden Methode als String übergeben. Dieses Argument wird ebenfalls von der JVM zur Verfügung gestellt. Diese Information wurde bereits über *invokedynamic* übergeben.
- *MethodType invokedType*: Hier wird die Signatur für die CallSite übergeben[Orab]. Dieses Argument wird ebenfalls von der JVM bereitgestellt, da die Signatur bereits über den Eintrag von *InvokeDynamic* im Constant-Pool bereitgestellt wurde. In unserem Beispiel wird hier *()Ljava/util/function/Consumer* übergeben.
- *MethodType samMethodType*: Über *samMethodType* wird die Signatur der zu implementierenden Methode des funktionalen Interfaces und dessen Rückgabewert angegeben. Bei unserem Beispiel wird ein Lambda-Ausdruck vom Typ des funktionalen Interfaces *Consumer* erwartet. Die Signatur der abstrakten Methode des Interfaces *Consumer.accept* ist *#30 (Ljava/lang/Object;)V*. Es wird genau ein Parameter vom Typ *Object* entgegen genommen. Als Rückgabewert wird V für *void* angegeben. Genau diese Signatur wird auch als Methodenargument übergeben, wie man oben bei den Methodenargumenten sehen kann.
- *MethodHandle implMethod*: Hier wird die aufzurufende Methode festgelegt. In unserem Beispiel wird dabei *invokestatic A.lambdafoo0:(Ljava/lang/String;)V*

übergeben. Es wird ein Method-Handle übergeben der auf die erzeugte statische Methode des Lambda-Ausdrucks referenziert.

- *MethodType instantiatedMethodType*: Die übergebene Signatur legt fest, welcher Typ während der Ausführung der Methode dynamisch erzwungen werden soll[Orab]. Dabei kann der Typ der gleiche wie bei *samMethodType* sein oder aber ein spezifischerer Typ. In unserem Beispiel wird *(Ljava/lang/String;)V* übergeben, da im Beispiel ein *Consumer* mit Typparameter *String* verwendet wird.

Wie bereits erwähnt kann sich die Übersetzungsstrategie für Lambda-Ausdrücke von Java-Version zu Version ändern. Java 8 unterstützt jene Strategie, bei der für jeden Lambda-Ausdruck dynamisch eine Klasse erzeugt wird, die sich wie eine innere Klasse verhält. Da dies aber zur Laufzeit geschieht, werden keine zusätzlichen *.class*-Dateien angelegt. Auf Grund dieser aktuell verwendeten Übersetzungsstrategie wird von der Lambda-Metafactory ein Objekt der Klasse *InnerClassLambdaMetafactory* instanziiert. Über dieses Objekt wird dann auch die CallSite mit Hilfe der Methode *buildCallSite* erzeugt.

CallSite

Im Zusammenhang mit Lambda-Ausdrücken wird eine *ConstantCallSite* verwendet[Orab]. Dabei handelt es sich, wie der Name schon sagt, um eine konstante CallSite. Dabei wird einmal bei der Erzeugung der CallSite ein Method-Handle übergeben. Dieser Method-Handle kann anschließend bei der konstanten CallSite nicht mehr geändert werden. Das bedeutet, dass der Befehl *invokedynamic*, welcher an eine konstante CallSite gebunden wird, permanent an die Zielmethode gebunden ist. Diese Zielmethode wird über den Method-Handle referenziert. Im Gegensatz dazu kann bei der herkömmlichen CallSite die referenzierte Methode jederzeit geändert werden.

In der Methode *buildCallSite*, welche letztendlich die CallSite der Metafactory zurückliefert, wird zuerst die Methode *spinInnerClass* ausgeführt. Hierbei wird das *Classfile* von der Java-Laufzeit erzeugt, welche das zum Lambda-Ausdruck gehörige, funktionale Interface implementiert. Die innere Klasse für den Lambda-Ausdruck wird dabei über die Methode *defineAnonymousClass* der Klasse *sun.misc.Unsafe* erzeugt. Dazu wird des Weiteren das Bytecode-Tool ASM verwendet[Kul07]. Die Existenz der damit erzeugten Klasse ist dem Classloader jedoch nicht bekannt[micd].

Bei der Erstellung der CallSite durch *buildCallSite* in der *InnerClassLambdaMetafactory* zeigt sich wieder die unterschiedliche Übersetzungsstrategie, je nach Kategorie des verwendeten Lambda-Ausdrucks. Ebenso wie bei der Umwandlung der Lambda-Ausdrücke durch den Compiler in eine private Methode, wird auch hier zwischen zustandslosen und jenen Lambdas, welche vom umliegenden Kontext Variablen erfassen, unterschieden.

Bei einem zustandslosen Lambda-Ausdruck wird eine *ConstantCallSite* mit einem konstanten Method-Handle zurückgegeben. Dieser Method-Handle wird über *MethodHandles.constant* erzeugt. Im Beispiel wird dabei als Typ das Interface *Consumer* angegeben

und als Objekt wird die durch *spinInnerClass* erzeugte innere Klasse, welche das funktionale Interface implementiert, angegeben. Der Method-Handle liefert bei der Ausführung immer die innere Klasse zurück. Bei einem Lambda-Ausdruck, der Variablenwerte von außerhalb übernimmt, wird stattdessen ein Method-Handle erzeugt, der auf den Konstruktor der inneren Klasse referenziert. Das bedeutet, hier wird jeweils ein neues Objekt für die innere Klasse instanziiert. Wobei bei einem zustandslosen Lambda-Ausdruck nur eine Instanz benötigt wird.

Zurück zum Bytecode unseres Beispiels. Über *astore_1* erfolgt nun, nachdem *invokedynamic* über die Bootstrap-Methode mit der CallSite gelinkt wurde, die Zuweisung der Implementierung des funktionalen Interfaces zur Variable *consumer*. Über *ldc* wird der konstante Wert, der String *"test"*, auf den Stack gepusht. Über *invokeinterface #4, 2* und der dazugehörigen Information im Constant-Pool InterfaceMethodref wird die implementierte Methode *accept* des Interfaces *Consumer* aufgerufen. Nachdem die CallSite einmal erzeugt wurde, kann der Lambda-Ausdruck immer über *invokeinterface* ausgewertet werden. Es wird also nicht erneut der gesamte Prozess der Verlinkung von *invokedynamic* benötigt. Das sieht man im folgenden Bytecode, wobei der *consumer.accept* beispielsweise drei mal in Folge ausgeführt wurde.

```
 0: invokedynamic #2,  0
 5: astore_1
 6: aload_1
 7: ldc           #3
 9: invokeinterface #4,  2
14: aload_1
15: ldc           #3
17: invokeinterface #4,  2
22: aload_1
23: ldc           #3
25: invokeinterface #4,  2
30: return
```

Über *invokedynamic* wird die mit Hilfe der Bootstrap-Methode die CallSite mit der auszuführenden Methode verlinkt. Anschließend kann der Lambda-Ausdruck ohne erneutes Verlinken mit *invokeinterface* ausgewertet werden.

Methoden-Referenz

Hier soll kurz auf die Übersetzung von Lambda-Ausdrücken, welche als Methoden-Referenz dargestellt werden, eingegangen werden. Dazu ändern wir den Lambda-Ausdruck aus dem vorherigen Beispiel in eine Methoden-Referenz um. Der Parameter wird direkt an *println* weitergereicht. Es muss also nicht extra eine anonyme Methode erzeugt werden.

```
public class A {
    public void foo() {
        Consumer<String> consumer = System.out::println
        consumer.accept("test");
    }
}
```

Hier betrachten wir nur die relevanten Stellen des JVM-Bytecodes. Daher wird nur der Bytecode der Methode *foo* angegeben. In diesem Fall wird auf eine bereits existierende Methode referenziert. Daher muss vom Compiler keine synthetische Methode angelegt werden. Zusätzlich dazu werden die durch *javap* hinzugefügten Kommentare für eine bessere Lesbarkeit abgekürzt. So wird beispielweise *java/lang* und *java/util* am Anfang von Verweisen weggelassen. Die Referenzen auf Konstanten werden dabei direkt aufgelöst.

```
public void foo();
  flags: ACC_PUBLIC
  Code:
    0: getstatic        Field System.out:PrintStream
    3: dup
    4: invokevirtual    Method Object.getClass:()Class
    7: pop
    8: invokedynamic    #0:accept:(PrintStream;)Consumer;
    13: astore_1
    14: aload_1
    15: ldc             String test
    17: invokeinterface Consumer.accept:(Ljava/lang/Object;)V
    22: return
```

Mit *getstatic* wird *System.out* vom Typ *PrintStream* auf den Stack geladen. Anschließend wird Wert am Stack dupliziert. Über *invokevirtual* wird die Methode *getClass* wird bei *System.out* aufgerufen. Die Bootstrap-Methode benötigt Zugriff auf diese Klasse und daher wird über *getClass* überprüft, ob die Klasse vollständig geladen wurde und Zugriff auf diese möglich ist[Fis15]. Da wir das Resultat der Methode nicht benötigen, wird dieses mit *pop* vom Stack geworfen. Anschließend folgt wie bei dem vorherigen Beispiel auch der Aufruf zu *invokedynamic*. Der restliche Bytecode ist genau wie im vorherigen Beispiel auch. Über *invokeinterface* wird *Consumer.accept* aufgerufen. Die Unterschiede sind hier, dass *invokedynamic* mit dem dynamischen Parameter *PrintStream* aufgerufen wird. Des Weiteren wird die Boostrap-Methode, statt mit einem Method-Handle, welcher auf die synthetische Lambda-Methode referenziert, mit einem Method-Handle auf die jeweilige Methoden-Referenz aufgerufen. In diesem Beispiel würde der Method-Handle folgendermaßen aussehen.

```
invokevirtual java/io/PrintStream.println:(Ljava/lang/String;)V
```

Capture von Werten in Lambda-Ausdrücken Hier wollen wir noch kurz die bereits angesprochenen Unterschiede zwischen zustandslosen und zustandsbehafteten Lambda-Ausdrücken und deren Umsetzung in JVM-Bytecode betrachten. Nachfolgend ein Lambda-Ausdruck der keine Werte vom umliegenden Kontext benötigt. Dies wollen wir anhand von kurzen Beispielen betrachten. Irrelevante Informationen werden für bessere Lesbarkeit weggelassen.

```
public void calculate() {
    IntBinaryOperator add = (int x, int y) -> x+y;
    add.applyAsInt(1,2);
}

public void calculate();
 0: invokedynamic
        applyAsInt:()Ljava/util/function/IntBinaryOperator;
 5: astore_1
 6: aload_1
 7: iconst_1
 8: iconst_2
 9: invokeinterface IntBinaryOperator.applyAsInt:(II)I
 14: pop
 15: return

private static int lambda$calculate$0(int, int);
 0: iload_0
 1: iload_1
 2: iadd
 3: ireturn
```

Hier kann der Lambda-Ausdruck als statische Methode umgesetzt werden. Als nächstes greifen wir auf den Wert einer lokalen Variable zu.

```
public void calculate() {
    int z=3;
    IntBinaryOperator add = (int x, int y) -> x+y+z;
    add.applyAsInt(1,2);
}

private static int lambda$calculate$0(int, int, int);
    0: iload_1
    1: iload_2
    2: iadd
    3: iload_0
```

```
    4: iadd
    5: ireturn
```

Hier wird der Wert der lokalen Variable als erster Parameter an die erzeugte Methode übergeben. Da es sich um eine lokale Variable handelt, wird auch hier eine statische Methode erzeugt. Als nächstes greifen wir auf eine Instanzvariable zu. Hierbei ist z außerhalb der Methode deklariert.

```
private int lambda$calculate$0(int, int);
    0: iload_1
    1: iload_2
    2: iadd
    3: aload_0
    4: getfield    Field z:I
    7: iadd
    8: ireturn
```

Da wir nun auf den umliegenden Kontext des Lambda-Ausdruckes zugreifen und es sich um Mitglieder der Klasseninstanz handelt, kann hier nur eine private Methode erzeugt werden. Dabei wird *this* als dynamisches Argument übergeben. Dies sehen wir beim Aufruf von *invokedynamic*:

```
invokedynamic
  applyAsInt:(LJavaMain;)Ljava/util/function/IntBinaryOperator;
```

Abschließend greifen wir innerhalb des Lambda-Ausdrucks auf eine statische Variable der umliegenden Klasse zu. Hierbei wird keine Referenz zur Klasse über *this* übergeben. Stattdessen kann über *getStatic* innerhalb des Lambda-Ausdrucks auf die statische Variable zugegriffen werden. Dadurch kann auch die synthetische Methode für den Lambda-Ausdruck wieder statisch umgesetzt werden.

```
private static int lambda$calculate$0(int, int);
0: iload_0
1: iload_1
2: iadd
3: getstatic    Field z:I
6: iadd
7: ireturn
```

Grundsätzlich können wir Lambda-Ausdrücke zu statischen Methoden übersetzen, außer sie benötigen Zugriff auf die umliegende Instanz des Objektes in dem sie deklariert wurden. Dazu gehören Zugriffe auf *this*, *super* und nicht-statische Klassenvariablen und Methoden.[Goe12b]

KAPITEL 6

Lambda-Ausdrücke in anderen objektorientierten Programmiersprachen

In diesem Kapitel werden Lambda-Ausdrücke und dazugehörige Konzepte und Umsetzungen anderer objektorientierter Programmiersprachen vorgestellt und teilweise mit der Umsetzung innerhalb von Java verglichen. Um einen Vergleich auf JVM-Basis durchzuführen, werden wir uns einige Konzepte von Scala genauer ansehen. Scala ist deshalb interessant, weil es eine Art Sonderstellung im Bezug auf objektorientiertes und funktionales Programmieren bietet. So sind einerseits alle Konstrukte der Sprache Objekte, andererseits sind aber auch sehr viele brauchbare funktionale Konstrukte innerhalb der Sprache einsetzbar. Dabei wird es im Kapitel *Performance* auch einen Vergleich zwischen Java und Scala in Hinsicht auf die Geschwindigkeit geben, jedoch wird die Performance innerhalb von Java im Vordergrund stehen. Außerhalb der JVM wird die Umsetzung funktionaler Konzepte bei drei weiteren Programmiersprachen betrachtet. So wird einerseits die Umsetzung bei den Java-ähnlichen Sprachen C-Sharp und C++, bei denen auch eher das objektorientierte Paradigma im Vordergrund steht, vorgestellt. Der Fokus ist in diesem Kapitel auf Scala gerichtet. In einem späteren Kapitel wird dann ein Vergleich der Konzepte zwischen Java und Haskell in Hinsicht auf funktionale Programmierung durchgeführt.

6.1 Scala

In diesem Unterkapitel soll die ebenfalls auf der JVM laufende Programmiersprache Scala näher betrachtet werden. Dabei ist vor allem die Umsetzung von Lambda-Ausdrücken auf Sprachebene und deren Übersetzung in Bytecode innerhalb der JVM interessant, da Scala die Lambda-Ausdrücke bereits vor Java als Sprachkonstrukt hinzugefügt hatte.

Hier wird unter anderem die Umsetzung der Lambda-Ausdrücke in JVM-Bytecode vorgestellt. Ein ebenso sehr interessanter Aspekt an Scala ist die Kombination aus objektorientiertem und funktionalem Paradigma. Die Sprache selbst orientiert sich stark an Java und C-Sharp, zumindest in Sachen Operationen, Kontrollstrukturen und teilweiser Übernahme des Typsystems beider Sprachen. Scala ist voll kompatibel zu Java. So können ohne Probleme Methoden aus Java-Klassen in Scala aufgerufen werden. Des Weiteren können Klassen in Scala von Klassen aus Java erben beziehungsweise Java-Interfaces implementieren. Dadurch kann beispielsweise auch Scala-Code in Java Frameworks verwendet werden.[OAC+04a]

Wie bereits im ersten Kapitel dieser Arbeit erwähnt wurde, wird in Scala alles als Objekt behandelt und jede Operation wird als Methodenaufruf umgesetzt, also als Nachrichten an die Objekte wie in Smalltalk[Gol84]. Dies wird durch folgendes Beispiel verdeutlicht. Bei der Operation $x + y$ wird diese intern als *x.+(y)* interpretiert. Es handelt sich beim Operator $+$ also um eine Methode, wobei der erste Operand x die Methode $+$ mit dem zweiten Operanden y als Parameter aufruft.[OAC+04a]

Bei der Deklaration von Variablen muss in Scala kein Typ angegeben werden. Dieser wird vom Compiler hergeleitet. Dazu kann entweder das Schlüsselwort *var* oder *val* verwendet werden. Dabei steht *var* für einen hergeleiteten Typen einer Variable. Dieser Wert kann anschließend auch geändert werden. Jedoch muss der neu zugewiesene Wert dem hergeleiteten Typen der Variable entsprechen. Im Gegensatz dazu kann mit *val* ein Wert definiert werden, der sich anschließend nicht mehr ändern lässt. Dies entspricht dem *final* Modifikator in Java. Da es bei der funktionalen Programmierung von Vorteil ist mit unveränderbaren Werten zu hantieren, eignet sich *val* sehr gut für funktionale Programmierung in Scala.

6.1.1 Klassen

Auch die Erstellung von Klassen unterscheidet sich etwas von der bekannten Java-Syntax. Dazu betrachten wir folgendes Beispiel.

```
class Punkt(xc: Int=0, yc: Int=0) {
  var x: Int = xc
  var y: Int = yc
}

def main(args: Array[String]) : Unit = {
  val punkt = new Punkt
  punkt.x = 2
  Predef.println(punkt.x, punkt.y)
}

//Ausgabe
(2,0)
```

Die Klasse soll einen Punkt repräsentieren, welcher aus einer X- und Y-Koordinate besteht. Die Variablen x und y sind vom Typ *Int*. Im Vergleich zu Java spart man in Scala einiges an Standard-Code bei der Erstellung von Klassen ein. Zum einen benötigen wir keinen Konstruktor für die Klasse *Punkt*. Dieser wird standardmäßig über die Angabe der Parameter direkt nach der Definition der Klasse über *class* in den runden Klammern festgelegt. Hier können auch default-Werte für die Parameter angegeben werden, in diesem Beispiel der Wert 0. Dadurch braucht man keine Werte für x und y bei der Erstellung eines neuen Punktes anzugeben. Des Weiteren kann man bei Scala bei Methoden die runden Klammern weglassen. Für die Variablen x und y werden keine *Getter* und *Setter* benötigt. Die Variablen ohne vorangehenden Modifikator sind standardmäßig *public*. Für öffentliche Variablen werden diese automatisch angelegt. Außerhalb kann man, wie man in *main* sieht, ganz normal auf x und y zugreifen. Man erspart sich hier also das typische *x.get()* und *x.set(2)*, das man von Java gewohnt ist.[OAC+04a]

Neben herkömmlichen Klassen, welche in Scala auch über *class* definiert werden, gibt es in Scala so genannte Objekte. Das zeigt sich hier am Beispiel des Lambda-Ausdrucks im nächsten Unterkapitel an Hand der Definition von *Main* mit *object*. Dabei handelt es sich um Klassen mit einer einzigen Instanz, welches auch als *Singleton* bezeichnet wird. Im Vergleich zu Java findet man hier bei der Definition der Methode *main* über das Schlüsselwort *def* zuerst die Parameter und erst am Ende den Rückgabewert der Methode. Der Typ *Unit* entspricht dem Typen *void* in Java. Vor der geschwungenen Klammer befindet sich auch noch optional das Zeichen =. Dadurch wird festgelegt, dass der Wert des letzten Ausdrucks der Methode von dieser Methode automatisch zurückgegeben wird. Dadurch erspart man sich die *return* Anweisung. Lässt man das Zeichen = weg, so impliziert dies den Rückgabewert *Unit*. Da der Rückgabewert einer Methode über Typinferenz vom Compiler bestimmt werden kann, kann dessen Angabe in der Methodensignatur entfallen[OAC+04b]. Typinferenz funktioniert jedoch nicht immer. Im Vergleich zu Java benötigen wir hier bei Parametern von Lambda-Ausdrücken einen Typ. In Java können die Typen über den Kontext des Lambda-Ausdrucks hergeleitet werden. Da wir in Scala jedoch meist den Typ des Rückgabewertes vom Compiler bestimmen lassen, müssen wir hier bei Parametern den Typ angeben. Auch bei einer rekursiven Funktion muss der Typ des Rückgabewerts explizit angegeben werden, da der Compiler den Typ nicht herleiten kann. [EPFb]

```
def fac(n: Int):Int = if( n == 0 ) 1 else n * fac(n-1)
```

Bei diesem Beispiel soll die Fakultät von n ausgerechnet werden. Dabei wird die Funktion *fac* solange mit *n-1* aufgerufen, bis n den Wert 0 erhält. Die Fakultät von n lässt sich über *n * (n-1) * (n-2) ... 1* ausrechnen. Hier muss man dem Compiler explizit den Typen des Rückgabewertes angeben.[EPFb]

6.1.2 Funktionen

In Scala sind Funktionen als *Bürger erster Klasse* in die Sprache eingebettet. Daher können die Funktionen den Variablen zugewiesen, in einer Datenstruktur wie beispielsweise einer Liste abgespeichert und als Parameter und Rückgabewert einer Methode verwendet werden. Es sind also ebenfalls Funktionen höherer Ordnung in Scala möglich. Zusätzlich dazu wird in Scala ebenso *Currying* von Funktionen unterstützt und auch die Definition von Funktionen innerhalb von Funktionen ist möglich (verschachtelte Funktionen). Beides ist auch in Java 8 mit Hilfe von Lambda-Ausdrücken realisierbar. Nun betrachten wir die Syntax von Lambda-Ausdrücken in Scala. Dabei können auch generelle Unterschiede im Bezug auf die Syntax aufgezeigt werden.[OAC+04a]

```
object Main {
  def main(args: Array[String]): Unit = {
    val f = (x:Int, y:Int) => x+y
    val result = f.apply(1,2)
  }
}
```

Die Syntax der Lambda-Ausdrücke in Scala ist jener in Java sehr ähnlich. In den runden Klammern befinden sich die Parameter, wobei zuerst der Name des Parameters und dann getrennt durch einen Doppelpunkt der dazugehörige Typ angegeben wird. Der Funktionskörper befindet sich nach dem typischen Pfeil, welcher hier im Gegensatz zu Java mit => dargestellt wird. Der Lambda-Ausdruck wird hier der Variable *f* zugewiesen. Interessant ist dabei, dass die Variable mit *val* deklariert wurde. Dadurch kann der Compiler den Typ durch Inferenz herleiten. Der Aufruf der Funktion erfolgt über die Methode *apply*. Es ist auch möglich, die Funktion über *f(1,2)* aufzurufen, da Operatoren in Scala überladen werden können. Des Weiteren benötigen wir keine geschwungenen Klammern beim Funktionskörper des Lambda-Ausdrucks. Wie man an diesem Beispiel sieht, wird in Scala im Vergleich zu Java kein ; am Ende der Zeile benötigt.

6.1.3 Funktionstypen

Die Frage, die sich nun stellt, ist, wie Lambda-Ausdrücke in Scala umgesetzt werden. Diese Frage ist deshalb besonders interessant, weil Scala bereits vor Java Lambda-Ausdrücke und Funktionen als *Bürger erster Ordnung* umsetzen konnte. Wie bereits erwähnt, war die Umsetzung in Java nicht ganz so einfach durchführbar, vor allem auf Grund der Rückwärtskompatibilität, die innerhalb der Java-Versionen eingehalten werden muss.

Scala unterstützt hier das Konzept der Funktionstypen. So kann die Signatur der Funktion, die durch den im vorherigen Beispiel verwendeten Lambda-Ausdruck repräsentiert wird, über den Funktionstyp wie folgt dargestellt werden[EPFb]. Dabei kann der Funktionstyp auch explizit vor dem Lambda-Ausdruck angegeben werden. Dann benötigt man

auch keine Typen bei den Parametern. Es ist auch möglich, die Syntax des Lambda-Ausdrucks mit expliziten Funktionstypen abzukürzen. Das nachfolgende Beispiel macht dies deutlich.[Ale13]

```
//Lambda-Ausdruck
(x:Int, y:Int) => x+y

//Funktionstyp
(Int, Int) => Int

//Lambda-Ausdruck mit explizitem Funktionstyp
val func: (Int,Int) => Int = (x,y) => x+y

//Verkürzte Variante, _ repräsentieren hier x und y
val func: (Int,Int) => Int = _ + _
```

Der Funktionstyp stellt also die Signatur der Funktion dar. Bei der Verwendung von *val* oder *var* brauchen wir uns über die genaue Signatur keine Gedanken machen, da die Parameter von uns Typen zugewiesen bekommen und der Rückgabewert vom Compiler berechnet wird. Verwendung finden die Funktionstypen beispielsweise bei der Definition von Funktionen höherer Ordnung.

```
def sort(array: Array[Int], order: (Int,Int) => Boolean) = {
  array.sortWith(order)
}

var z = Array(3,1,2)
z = sort(z, (x:Int, y:Int) => x<y )
```

Hier definieren wir eine Funktion *sort*. Diese nimmt ein Int-Array entgegen und eine Funktion. Die Funktion, die als zweiter Parameter entgegen genommen wird, hat den Funktionstypen *(Int,Int) => Boolean*. Die Methode *sortWith* benötigt als einzigen Parameter eine Funktion mit selbem Funktionstypen. Für diese Funktion werden zwei Parameter vom Typ *Int* als Eingabeparameter verwendet und als Resultat erhalten wir einen Wert vom Typ *Boolean* zurück. Um eine aufsteigende Sortierung zu erhalten, übergeben wir also den Lambda-Ausdruck mit dem Funktionskörper $x<y$. Es handelt sich bei *sort* und *sortWith* um Funktionen höherer Ordnung. Die Signatur beziehungsweise der Funktionstyp von *sort* würde folgendermaßen aussehen.

```
(Array[Int],(Int,Int) => Boolean) => Array[Int]
```

6.1.4 Umsetzung der Lambda-Ausdrücke in Scala

Zurück zu unserem Lambda-Ausdruck *(x:Int, y:Int) => x+y* mit Funktionstyp *(Int, Int) => Int*. Wie schon erwähnt wurde, handelt es sich bei Funktionen in Scala um Objekte. Des Weiteren werden auf Ebene der JVM keine Funktionstypen unterstützt. Daher werden Funktionstypen als syntaktischer Zucker bereitgestellt. Die Funktionstypen stehen hier für Typklassen. So handelt es sich bei unserem Beispiel um folgende Typklasse.[EPFa]

```
trait Function2[-T1, -T2, +R] {
  def apply(v1: T1, v2: T2): R
}
```

Traits sind mit Interfaces in Java vergleichbar. In unserem Beispiel würde der Lambda-Ausdruck folgender Instanz des Traits *Function2* entsprechen. Da es sich bei *apply* um die einzige abstrakte Methode von *Function2* handelt, reicht folgende Definition aus. Über *apply* wird die Funktion festgelegt. Beim Aufruf der Methode wird aus *f(x,y)* *f.apply(x,y)*.[OAC+04a]

```
new Function2[Int, Int, Int] {
  def apply(x: Int, y: Int): Int = x+y
}
```

Über diesen Trait werden an die Funktionen noch drei konkrete Methoden vererbt. Interessant ist vor allem die Methode *curried*. Dadurch kann die Funktion über einzelne Argumente aufgerufen werden. Somit kann auch eine partielle Anwendung der Funktionen erfolgen.[EPFa]

Erfassung von Variablen

Auch hier wollen wir wieder die Erfassung von Variablen des umliegenden Kontextes überprüfen. Auch in Scala handelt es sich bei Lambda-Ausdrücken um ein Closure. Daher können Variablen vom umliegenden Kontext für den Lambda-Ausdruck erfasst werden und innerhalb des Funktionskörpers verwendet werden. Im Gegensatz zu Java ist die Verwendung von Variablen nicht auf *effektiv finale* Variablen beschränkt. Man kann also innerhalb der Lambda-Ausdrücke ganz normal auf die Variablen des umliegenden Kontextes zugreifen, unabhängig davon, ob sich deren Wert ändern darf oder nicht. Des Weiteren ist es auch erlaubt, den Wert der äußeren Variablen innerhalb des Lambda-Ausdrucks zu verändern. Besser ist es hier, im Sinne *korrekter* funktionaler Programmierung auf Seiteneffekte zu verzichten und daher nur mit *val* deklarierte Werte zu verwenden. Dadurch bleiben die Lambda-Ausdrücke Thread-Safe und können in einer parallelen Programmierumgebung eingesetzt werden. Innerhalb des Lambda-Ausdrucks können wie in Java neue Variablen angelegt werden. Diese sind im umliegenden Kontext nicht verfügbar. Auch der Zugriff auf *this* wird wie in Java gehandhabt.

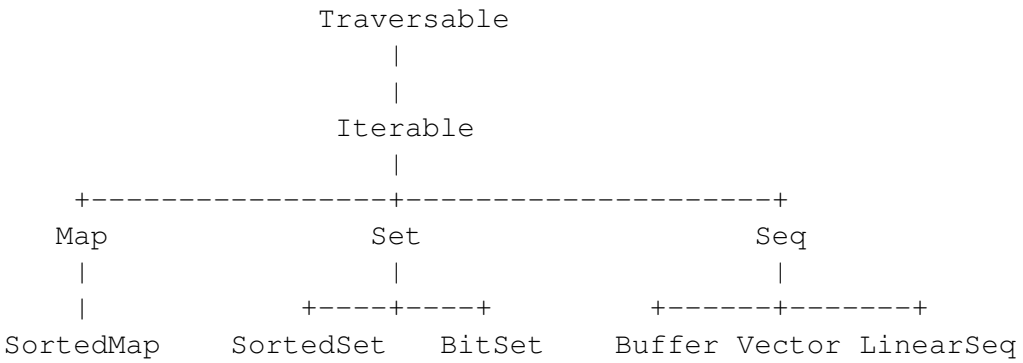
```
var a = 2
val f = (x:Int, y:Int) => { x+y+a; }

Predef.println(f(2,2))
a=3
Predef.println(f(2,2))
```

Der Zugriff auf Variablen des umliegenden Kontextes werden über eine Referenz auf die Variable durchgeführt. Daher erhalten wir nach Veränderung der Variable *a* ein anderes Ergebnis für den Aufruf von *f(2,2)*. Ein weiterer Grund für die Vermeidung solcher Szenarien ist referentielle Transparenz, welche für die funktionale Programmierung ebenfalls ein wichtiges Konzept ist. Der Aufruf von *f* mit denselben Parametern sollte immer das gleiche Ergebnis zurückliefern. Dadurch kann mathematisch besser darüber geurteilt werden und Optimierungen können besser durchgeführt werden. So können Funktionsaufrufe durch ihr Ergebnis ersetzt werden.[Ale13]

6.1.5 Scala Collections

Hier wollen wir kurz die Collections API von Scala vorstellen. Diese wurde in Version 2.8 erneuert, welche im Jahr 2008 veröffentlicht wurde. Diese befinden sich im Paket *scala.collection*. Bei Scala wird zwischen veränderbaren(*scala.collection.mutable*) und unveränderbaren(*scala.collection.immutable*) Collections unterschieden. Nachfolgend sehen wir den Aufbau der Grundklassen für die Collections API[Ode09]:



Dabei ist über *Traversable* und *Iterable* die Methode *forEach* definiert, die alle Elemente der Collection durchläuft und eine Funktion darauf anwendet. Das bedeutet, die Collections in Scala unterstützen Funktionen höherer Ordnung. Hier werden Funktionen wie beispielsweise *map*, *flatMap* und *filter* direkt zugänglich gemacht. Wir müssen also nicht wie in Java die Collection in einen Stream umwandeln. Des Weiteren gibt es wie in Java Collections für Maps, Sets und Sequenzen - wobei Seq in etwa der dem Java Interface *List* entspricht. *LinearSeq* enthält unter anderem *List* und *Stream*, wobei beide unveränderliche Datenstrukturen sind. Diese sind für schnelle Zugriffe geeignet. Jedes

Element von *List* und *Stream* ist dabei über *head*, welches das aktuelle Element der Liste darstellt, und dem Rest der Liste *tail* definiert. Vektoren hingegeben, wie zum Beispiel Arrays sind besser für das Hinzufügen von neuen Elementen oder Änderungen bestehender Elemente geeignet. Wie bei Java Streams können Scala Collections über *.range(start, end)* generiert werden.[Ode09]

Scala Streams

In Scala hat ein Stream nicht dieselbe Bedeutung wie in Java, da wir wie bereits erwähnt direkt auf die Collections Funktionen höherer Ordnung anwenden können. In Scala verhält sich ein Stream wie eine Liste. Die einzigen Unterschiede sind, dass die Elemente des Streams verzögert ausgewertet werden. Daher sind unendliche Streams möglich. Anders als in Java kann man einen Stream hier schon als Collection betrachten in der Elemente abgespeichert sind. Sobald die Elemente verzögert ausgewertet wurden, werden diese nämlich in einem Cache zwischengespeichert. Das wird in der funktionalen Programmierung auch als Memoization[HM97] bezeichnet. Es gibt keine parallelen Streams in Scala. Stattdessen werden parallele Collections verwendet. Dessen Aufbau ist im nächsten Unterkapitel genauer beschrieben. Folgendes Beispiel soll die Funktionsweise von Streams in Scala veranschaulichen.

```
val stream = Stream.range(1, 100).map(x => {
    println("EVALUATE")
    x
})

stream.take(4).toList
stream.take(5).toList
```

Wir erzeugen in Scala eine Liste, die verzögert ausgewertet wird, repräsentiert durch einen *Stream* mit Werten zwischen eins und hundert. Über die Operation *map* lassen wir uns *EVALUATE* in der Konsole ausgeben, sobald der Wert von x berechnet wird. Hier geben wir lediglich x zurück. Der Typ des Streams ist *Stream(1, ?)* da lediglich der Kopf des Streams sofort ausgewertet wird. Der Rest wird erst berechnet wenn die Werte benötigt werden. Daher ist ein Stream nicht zu hundert Prozent lazy. Wenn wir nun die ersten vier Elemente des Streams in eine Liste umwandeln, so erhalten wir in der Konsole viermal die Ausgabe von *EVALUATE*. Der Stream liefert uns nun als Ausgabe *Stream(1, 2, 3, 4, ?)*, da die ersten vier Elemente ausgewertet wurden und der Rest des Streams noch unbekannt ist. Dies ist durch das Fragezeichen gekennzeichnet. Danach lassen wir uns die ersten fünf Elemente ausgeben. Daber wird nur einmal *EVALUATE* ausgegeben. Die Ausgabe des Streams ergibt nun *Stream(1, 2, 3, 4, 5, ?)*. Die gesamte Auswertung des Streams können wir mit *force* erzwingen.

Parallele Collections

Da in Scala ein Stream eine andere Bedeutung als in Java hat, werden parallele Berechnungen und Operationen nicht über parallele Streams angeboten. Stattdessen verwendet Scala parallele Collections. Diese wurden in Scala 2.9 im Jahr 2011, also etwa ein Jahr nach der Überarbeitung der Collections API veröffentlicht. Für die verschiedenen Collections gibt es dabei ein paralleles Gegenstück. Mit Hilfe der Methode *par* kann eine sequenzielle in eine parallele Collection umgewandelt werden. Dabei geht es bei den parallelen Collections jedoch nicht um nebenläufige Collections, auf welche synchronisiert von mehreren Threads aus zugegriffen werden soll. Stattdessen verwenden Scala hier denselben Ansatz wie Java bei den parallelen Streams. Diese bieten die Möglichkeit an, sich in Teile aufzusplitten und am Ende die Ergebnisse wieder zusammenzuführen. Dies geschieht in Scala über *Splitter* und *Combiner*. Als nächstes wollen wir Java Spliteratoren mit Scala Splitters vergleichen. Dazu betrachten wir nachfolgend die Definition des Splitters in Scala.

```
trait Splitter[T] extends Iterator[T] {
    def split: Seq[Splitter[T]]
}
```

In Scala erweitert der Splitter den Iterator. Der Iterator ist in Scala ähnlich wie in Java aufgebaut und besitzt die Methoden *hasNext* und *next*. Dadurch bleibt in Scala der Overhead des Iterators erhalten, da wir immer hasNext vor next aufrufen müssen. Zusätzlich definiert der Splitter eine Methode um sich selbst in mehrere Teile aufzuteilen. Hier sehen wir auch schon den nächsten Unterschied im Vergleich zum Spliterator. Der Spliterator in Java liefert bei der Methode *trySplit* nämlich nur einen neuen Spliterator zurück. In Scala wird eine Sequenz von Splitter-Elementen zurückgeliefert. Der ursprüngliche Splitter wird nach Anwendung von *split* ungültig. Im Hintergrund verwenden parallele Collections wie in Java das bereits vorgestellte Fork-Join-Framework, zusammen mit dem Fork-Join-Thread-Pool. Die Operationen werden als Tasks des Fork-Join-Frameworks umgesetzt.[PBRO11]

```
class Map[S](f: T => S, s: Splitter[T]) extends Task {
    var cb = newCombiner
    def split = s.split.map(subspl => new Map[S](f, subspl))
    def leaf() = while (s.hasNext) cb += f(s.next)
    def merge(that: Map[S]) = cb = cb.combine(that.cb)
}
```

Split teilt das Problem in kleinere Teilprobleme auf. Die Methode *leaf* durchläuft die Elemente des Splitters sequenziell und setzt die Mapping-Operation um. Anschließend fügt *merge* die Ergebnisse zusammen.

Scala Collection Views

Scala Collections werden grundsätzlich strikt ausgewertet[PBRO11]. Wenn wir also mehrere Operationen aneinanderhängen, werden immer Zwischenresultate in Form von Collections erzeugt. Dadurch wird Speicherplatz und Rechenzeit verschwendet, wenn wir nicht alle Zwischenresultate benötigen. Die einzige Collection die nicht strikt ausgewertet wird, ist der bereits vorgestellte *Stream*. Anhand des folgenden Beispiels wollen wir uns die strikte Auswertung in Scala ansehen.

```
List.range(0,10).filter(x => {
    println("FILTER")
    x%2 ==0
}).map(x => {
    println("MAP")
    x*2
}).take(2)
```

Um die Reihenfolge der Ausführung besser nachvollziehen zu können, geben wir in den filter und mapping Operationen jeweils eine Zeile in der Konsole aus. Als Ergebnis erhalten wir hier, nicht wie bei Java Streams abwechselnd die Ausgabe von *FILTER* und *MAP*. Stattdessen wird hier zuerst zehnmal *FILTER* ausgegeben. Dabei wir eine neue Liste aus den gefilterten Werten erzeugt. Erst danach wird fünfmal *MAP* ausgegeben, obwohl wir über *take(2)* nur zwei Elemente benötigen würden.

Um hier das Problem der strikten Auswertung innerhalb von Scala Collections zu umgehen, können wir so genannte *Views* verwenden. Dazu schreiben wir nach der Methode *range(0,10)* einfach *.view()* dazu. Dadurch erhalten wir eine verzögerte Auswertung wie bei Java Streams. Da es sich bei *take* um keine terminierende Operation handelt, erhalten wir keine Ausgabe. Wenn wir nun eine terminierende Operation, wie *sum* verwenden, erhalten wir als Ausgabe *FILTER MAP FILTER FILTER MAP*. Anstelle einer terminieren Operation können wir über *force* die Umwandlung von einer View zu einer Collection in Gang setzen. Der Unterschied zwischen Views und Streams in Scala ist, dass die Ergebnisse des Streams zwischengespeichert werden. Bei der View hingegen werden alle Berechnungen jedes Mal neu ausgewertet.

6.1.6 JVM-Bytecode von Lambda-Ausdrücken in Scala

Hier wollen wir uns den Aufbau von Lambda-Ausdrücken in Scala in Form von JVM-Bytecode genauer anschauen. Dafür betrachten wir wie im Kapitel über die JVM den JVM-Bytecode mit Hilfe von *javap*. Als Beispiel nehmen wir einen Lambda-Ausdruck für die Addition zweier Werte.

```
Lambda.scala in der Methode main:
val add = (x: Int, y: Int) =>  x+y;
add(2,3);
```

Scala 2.11 Betrachten wir zunächst den JVM-Bytecode der entsteht, wenn wir den Code mit Scala 2.11 kompilieren. Dabei erzeugt der Scala-Compiler drei *.class* Dateien.

```
Lambda.class
Lambda$.class
Lambda$$anonfun$1.class
```

Da wir unsere Klasse nicht über *class* sondern als *object* erstellt haben, werden die beiden *class* Dateien *Lambda.class* und *Lambda$.class* erzeugt. Dabei wird unsere Klasse als *Singleton* erzeugt. Nachfolgend betrachten wir zuerst *Lambda.class*.

```
0: getstatic Field Lambda$.MODULE$:LLambda$;
3: aload_0
4: invokevirtual Method Lambda$.main:([Ljava/lang/String;)V
7: return
```

Hier wird eigentlich nur die Methode *main* unserer Singleton-Klasse aufgerufen. Deren Code ist in *Lambda$.class* definiert. Hier schauen wir uns nur die Methode *main* an.

```
0: new           class Lambda$$anonfun$1
3: dup
4: invokespecial Method Lambda$$anonfun$1."<init>":()V
7: astore_2
8: aload_2
9: iconst_2
10: iconst_3
11: invokeinterface scala/Function2.apply$mcIII$sp:(II)I
16: pop
17: return
```

Hier wird eine neue Instanz der Klasse *Lambda$$anonfun$1* erzeugt. Diese repräsentiert unseren Lambda-Ausdruck. Eine Referenz des erzeugten Objekts landet auf dem Stack. Mit *dup* wird diese Referenz am Stack dupliziert. Der JVM-Befehl *invokespecial* initialisiert die Lambda-Klasse. Dabei wird die Objektreferenz vom Stack genommen, daher wird vorher *dup* aufgerufen. *astore_2* speichert den Lambda-Ausdruck in die lokale Variable und *aload_2* nimmt diesen und legt ihn wieder auf den Stack. Zusätzlich dazu kommen die Werte 2 und 3 auf den Stack um die Berechnung letztlich mit *invokeinterface* durchzuführen. Nachfolgend sehen wir den JVM-Bytecode von *Lambda$$anonfun$1*.

```
0: aload_0
1: iload_1
```

```
2: iload_2
3: invokevirtual Method apply$mcIII$sp:(II)I
6: ireturn
```

Hier wird lediglich die apply Methode des Interfaces aufgerufen. In Scala 2.11 werden Lambda-Ausdrücke daher direkt als innere Klassen umgesetzt, wie in Java 7. Dadurch sind die Lambda-Ausdrücke hier nur syntaktischer Zucker im Gegensatz zu Java 8.

Scala 2.12 Als nächstes sehen wir uns die Umsetzung in der neueren Version Scala 2.12 an. Dieses mal werden beim Kompilieren mit Scala 2.12 nur zwei Dateien erzeugt.

```
Lambda.class
Lambda$.class
```

Hier wird also keine eigene *class* Datei für den Lambda-Ausdruck angelegt. Betrachten wir die Methode *main* in *Lambda$.class*.

```
0: invokedynamic
    apply$mcIII$sp:()Lscala/runtime/java8/JFunction2$mcIII$sp;
5: astore_2
6: aload_2
7: iconst_2
8: iconst_3
9: invokeinterface scala/Function2.apply$mcIII$sp:(II)I
14: pop
15: return
```

Der Lambda-Ausdruck wird hier, wie in Java 8, über *invokedynamic* an die *CallSite* gebunden. Mit *invokeinterface* wird die apply Methode von *Function2* aufgerufen. Dieses verwendet ebenfalls die Annotation *@FunctionalInterface*. Der Lambda-Asdruck wird dabei als öffentliche statische Methode innerhalb unserer Klasse umgesetzt.

```
public static final int $anonfun$main$1(int, int);
0: iload_0
1: iload_1
2: iadd
3: ireturn
```

Die beiden Parameter werden auf den Stack geladen, addiert und das Ergebnis wird zurückgeliefert. Was passiert nun, wenn wir aus dem umliegenden Kontext auf Instanz-Variablen zugreifen. Wie bei der Java-Umsetzung von Lambda-Ausdrücken werden wir jetzt auf eine lokale Instanzvariable innerhalb des Lambda-Ausdrucks zugreifen. Dabei wird folgender JVM-Bytecode erzeugt.

```
val add = (x: Int, y: Int) =>  x+y+this.z
```

```
public static final void $anonfun$main$1(Lambda, int, int);
```

Im Gegensatz zu Java, wo eine private nicht-statische Methode erzeugt wurde, wird in Scala vom Compiler auch in diesem Fall eine öffentliche statische Methode erzeugt. Scala verwendet hier wie in Java als Bootstrapmethode für *invokedynamic* die LambdaMetaFactory.

```
BootstrapMethods: invokestatic
    java/lang/invoke/LambdaMetafactory.altMetafactory
```

Für mehr Kontrolle darüber, wie die Lambda-Ausdrücke umgesetzt werden, wird hier jedoch statt der Methode metaFactory die Methode *altMetafactory* verwendet. Dazu gehört die Möglichkeit, die Lambda-Ausdrücke serialisierbar zu machen. Davon wird bei der Umsetzung in Scala Gebrauch gemacht. Ansonsten sind die Methoden identisch.[Orab]

6.2 Lambdas in C-Sharp

In diesem Kapitel wollen wir die Umsetzung von Lambda-Ausdrücken in einer anderen weit verbreiteten, objektorientierten Programmiersprache betrachten. Um auch einen Vergleich außerhalb der JVM zu erhalten, werden wir hier die Umsetzung von funktionalen Programmierkonzepten innerhalb der Sprache C-Sharp näher betrachten und diese anschließend in Relation zur Umsetzung in Java setzen. Dabei soll zuerst kurz erwähnt werden, wann die jeweiligen Konstrukte zur Sprache hinsichtlich Version und Erscheinungsdatum hinzugefügt wurden. Anschließend wird überblicksmäßig die Syntax von Lambda-Ausdrücken in C-Sharp erläutert. Danach sollen Limitierungen bei der funktionalen Programmierung in C-Sharp erkannt werden. Des Weiteren soll ein kurzer Vergleich zwischen Java Streams und C-Sharp LINQ (Language Integrated Query) durchgeführt werden.

Zuerst betrachten wir die funktionalen Ansätze innerhalb von C-Sharp und dessen Weiterentwicklung im Bezug auf die verschiedenen Versionen. In C-Sharp Version 1.0 wurden Funktionen als *Bürger erster Klasse* zur Programmiersprache hinzugefügt. Dies geschah über das Konstrukt des so genannten Delegaten. Diese konnten Anfangs nur durch Methoden, die bereits über einen Namen deklariert wurden, festgelegt werden. Dieses Konzept wird im Anschluss genauer erklärt. C-Sharp 1.0 wurde im Jahr 2002 veröffentlicht. In Version 2.0 wurden unter anderem anonyme Methoden hinzugefügt. Dadurch war es möglich Delegaten direkt zu erstellen. Des Weiteren wurden auch Iteratoren zur Sprache hinzugefügt. In Version 3.0, welche zusammen mit *.NET 3.5* im April 2010 veröffentlicht wurde, wurden Lambda-Ausdrücke zu C-Sharp hinzugefügt. Hinzu kamen Typinferenz für lokale Variablen und Lambda-Ausdrücke. Als weiterer Verwendungszweck für Lambda-Ausdrücke wurden Expression-Trees ebenfalls zu Version 3.0 hinzugefügt.

Delegaten bezeichnen einen Typ, welcher eine Referenz auf eine Methode darstellt. Der Typ des Delegaten wird über dessen Namen festgelegt. Delegaten können auch als Methoden-Signaturen angesehen werden. Dabei werden die Typen der Parameter und der Typ des Rückgabewertes angegeben. Dem Delegaten kann anschließend eine beliebige Methode zugewiesen werden. Dabei muss jedoch die Signatur, also die Typen der Eingangsparameter und der Typ des Rückgabewertes mit jenen des Delegaten kompatibel sein. Beim Aufruf des Delegaten wird der Methodenaufruf an die jeweils zugewiesene Methode weitergereicht.[Mica]

Anschließend kann die jeweilige Methode über den Delegaten aufgerufen werden. Damit bilden Delegaten den Grundbaustein, um Methoden an andere Methoden weiterzugeben beziehungsweise zurückzuliefern. Somit werden Funktionen höherer Ordnung ohne größere Umstände ermöglicht. Die Referenz auf die Methode kann zur Laufzeit geändert werden. Delegaten entsprechen somit weitgehend funktionalen Interfaces in Java. Wenn man einen Vergleich zwischen Delegaten und Java vor Version 8 herstellen möchte, so könnte man diese mit SAM-Typen vergleichen, welche bereits im Kapitel *Lambda-Ausdrücke* vorgestellt wurden. Der Vorteil von Delegaten besteht darin, dass nicht extra Interfaces und Klassen erzeugt werden müssen.[Mica]

Des Weiteren sind Delegaten Funktionszeigern in C++ ähnlich. Sie sind jedoch im Vergleich zu diesen typsicher. Delegaten können ohne weiteres mehrere Methoden miteinander verketten. So können bei dessen Ausführung mehrere Methoden über einen Aufruf ausgeführt werden. Wie wir bereits gesehen haben, gibt es auch bei einigen funktionalen Interfaces von Java die Möglichkeit, ebenfalls mehrere Funktionen mit Hilfe der Methode *andThen* oder *compose* aneinander zu hängen.

Nachfolgend wird die Syntax von Delegaten vorgestellt. An Hand eines Beispiels soll auch der Aufruf mehrerer verketteter Methoden dargestellt werden.

```
public delegate return_typ delegate_name(argumente_mit_typ)
```

Hier sieht man die allgemeine Syntax für Delegaten in C-Sharp. Als Beispiel wollen wir nun einen Delegaten für eine Funktion mit zwei Parametern erzeugen welche beide vom Typ *int* sind. Diese Funktion stellt also eine binäre Operation dar. Der Rückgabewert ist ebenfalls vom Typ *int*.

```
public delegate int Intfunction(int a, int b)
```

Nachdem wir nun eine Signatur für eine binäre Operation, welche auf zwei Operanden vom Typ *int* angewandt werden soll, mit Hilfe eines Delegaten festgelegt haben, können wir diesem Delegaten die gewünschte Methode zuweisen. Im folgenden Beispiel werden wir zwei Methoden festlegen. Diese werden anschließend dem Delegaten zugewiesen. Die Methode *Add* repräsentiert dabei eine Addition und *Multiply* berechnet das Produkt zweier Operanden. Um die Verkettung der Methodenaufrufe durch Delegaten in diesem

Beispiel besser darstellen zu können, wird innerhalb der Methoden *Add* und *Multiply* die Methode *Console.WriteLine* aufgerufen, um das Ergebnis der jeweiligen Funktion am Terminal auszugeben.

```
public delegate int BinaryIntFunction(int a, int b);

public static int Add(int x, int y)
{
    Console.WriteLine("Add: " + (x + y));
    return x + y;
}

public static int Multiply(int x, int y)
{
    Console.WriteLine("Multiply: " + x * y);
    return x * y;
}

static void Main(string[] args)
{
    BinaryIntFunction func = Add;
    func(2, 3);

    func = Multiply;
    func(2, 3);

    func += Add;
    func(2, 3);

    func -= Multiply;
    func(2, 3);
}
```

In diesem Beispiel weisen wir dem Delegaten *BinaryIntFunction* zuerst die Methode *Add* für die Addition zu. Anschließend kann die Methode über den Delegaten aufgerufen werden. Hier erhalten wir das Ergebnis 5. Anschließend sehen wir, wie man während der Laufzeit auch andere Methoden zuweisen kann. Dazu wird nun die statische Methode *Multiply* dem Delegaten *func* zugewiesen. Am Ende der *Main*-Methode sieht man ein Beispiel für die Verkettung von Methoden, welcher auch als Multicall bezeichnet wird. Dazu wird mit dem Operator += dem Delegaten *func*, welcher zu diesem Zeitpunkt auf die statische Methode *Multiply* referenziert, eine weitere Referenz auf die Methode *Add* hinzugefügt. Dadurch wird beim anschließenden Aufruf *func(2, 3)* zuerst die Multiplikation und anschließend die Addition durchgeführt. Mit Hilfe des Operators -= können so auch

wieder Methoden vom Delegaten entfernt werden. Dadurch erhält man bei dem letzten Aufruf von *func* mit den Argumenten 2 und 3 lediglich das Ergebnis der Addition. In Java 8 würde dieser Delegat dem funktionalen Interface *IntBinaryOperator* aus dem package *java.util.function* entsprechen. Dieses Interface ist folgendermaßen definiert.

```
@FunctionalInterface
public interface IntBinaryOperator {
    int applyAsInt(int left, int right);
}
```

In dem vorherigen Beispiel wurden bereits vorher festgelegte Methoden den Delegaten zugewiesen. Ab C-Sharp Version 2.0 war es dann möglich, einen Delegaten über eine anonyme Methoden zu instanziieren. Diese Vorgehensweise lässt sich mit anonymen inneren Klassen von Java vergleichen. So wäre die Addition aus dem vorherigen Beispiel mit Hilfe von anonymen Methoden folgendermaßen umzusetzen.[Mica]

```
public delegate int BinaryIntFunction(int a, int b);
BinaryIntFunction func = delegate (int a, int b) { return a+b };
```

Der Typ des Rückgabewertes muss bei der Definition des Delegaten nicht angegeben werden. Der Rückgabewert muss jedoch kompatibel zur Signatur des Delegaten sein, anderenfalls wird eine Fehlermeldung ausgegeben. Die Angabe des Typs innerhalb der Parameterliste ist nach wie vor erforderlich. In Version 3.0 wurden Lambda-Ausdrücke zu C-Sharp hinzugefügt.[Mica]

```
public delegate int BinaryIntFunction(int a, int b);
BinaryIntFunction func = (int x, int y) => { return x + y; };
```

Die Syntax unterscheidet sich kaum im Vergleich zu Java. Auf der linken Seite des charakteristischen Pfeils des Lambda-Ausdrucks befindet sich die Parameterliste. Rechts davon findet man den Funktionskörper. Der Pfeil wird im Gegensatz zu Java mit => statt mit -> angegeben. Auf Grund von Typinferenz kann man bei den Parametern den Typ weglassen. Befindet sich im Funktionskörper nur ein Ausdruck, so kann man auf die geschwungenen Klammern und auf das *return*-Statement verzichten, wie man im folgenden Beispiel für die Addition sehen kann. Verwendet man nur einen Parameter, so kann man wie in Java die runden Klammern weglassen.

```
public delegate int BinaryIntFunction(int a, int b);
BinaryIntFunction func = (x, y) => x + y;
```

Um nicht jedes mal einen eigenen Delegaten definieren zu müssen, gibt es einen generischen Delegaten mit der Signatur *Func<T, TResult>*. Dabei werden Funktionen mit bis zu 16 Eingabeparametern unterstützt. Somit lässt sich der Lambda-Ausdruck für die Addition dem generischen Delegaten *Func<T1, T2, TResult>* zuweisen.

```
Func<int,int,int> test = (x,y) => x + y;
```

6.2.1 Erfassung der Variablen

Auch in C-Sharp darf auf Variablen, welche außerhalb des Lambda-Ausdrucks liegen, zugegriffen werden, sofern sich die jeweilige Variable im Geltungsbereich der umliegenden Methode des Lambda-Ausdrucks befindet. Solche Variablen bleiben für den Lambda-Ausdruck zwischengespeichert, auch dann wenn normalerweise der *Garbage-Collector* die Variable bereits freigeben würde. Von Lambda-Ausdrücken erfasste Variablen können dann vom Garbage-Collector freigegeben werden, sofern der zugehörige Lambda-Ausdruck ebenfalls nicht mehr benötigt wird. Ebenso wie in Java können innerhalb der Lambda-Ausdrücke neue Variablen definiert werden. Diese sind außerhalb nicht sichtbar.

Wie wir in Hinsicht auf die Geltungsbereiche von Variablen in Java bereits gesehen haben, ist es dort nur möglich auf effektiv finale Variablen außerhalb des Lambda-Ausdruckes zugreifen zu können. In C-Sharp gilt diese Einschränkung hingegen nicht. Man kann also veränderbare Variablen innerhalb eines Lambda-Ausdrucks einfangen. Ebenso ist es möglich, die innerhalb des Funktionskörpers des Lambda-Ausdrucks erfassten Variablen zu manipulieren. Somit wäre folgendes Beispiel möglich.[Mica]

```
delegate void D(int i);
delegate bool D2(int i);

public void Test() {
    int j = 0;
    D del = (x) => { j=x; };
    D2 del2 = (x) => { return x==j };

    //liefert false
    del2(10);
    del();
    //liefert true
    del2(10);
}
```

Man könnte also beispielsweise über einen Lambda-Ausdruck, der einem Delegaten zugewiesen wird, den Wert einer Variable der umliegenden Methode, in diesem Fall *j*, verändern. Durch diesen Seiteneffekt, der beim Aufruf des Lambda-Ausdrucks eintritt,

wird jedoch die parallele Verarbeitung auf Grund eines gleichzeitigen Variablenzugriffs erschwert. Man würde hier zusätzlich einen Synchronisationsmechanismus benötigen. Der Code ist daher nicht mehr *Thread-Safe*. Dadurch ist die Einschränkung im Bezug auf Java nicht als direkter Nachteil hinsichtlich effektiver funktionaler Programmierung zu sehen. Dadurch wird eine Kategorie von Seiteneffekten vermieden und es wird einfacher möglich funktionalen Code zu programmieren welcher ohne Probleme parallel abgearbeitet werden kann.

6.2.2 LINQ

Hier soll ein kurzer Vergleich zwischen der Streams API von Java und der so genannten Language-Integrated Query (LINQ) von C-Sharp vorgestellt werden. Genauer gesagt handelt es sich bei LINQ um ein Framework, welches von Microsoft im Zusammenhang mit *.NET* Version 3.5 veröffentlicht wurde. Es wurde gemeinsam mit C-Sharp 3.0 im November 2007 herausgebracht. Das *.NET-Framework* bildet dabei die Entwicklungsplattform für Sprachen wie C-Sharp, Visual Basic oder F-Sharp. So werden unter anderem der Compiler und die Laufzeit für C-Sharp von *.NET* zur Verfügung gestellt[Micc]. Ziel von LINQ ist es, dem Programmierer einen einheitlichen Zugriff auf Daten von unterschiedlichsten Quellen zu geben. Darunter beispielsweise XML-Dateien, SQL-Datenbanken und Zugriff auf Datenstrukturen innerhalb von C-Sharp. Im Vergleich mit der Streams API sind für uns nur Operationen auf Elementen einer Collection relevant. Interessant ist hier jedoch zumindest die Anmerkung, dass es über LINQ direkt eine API zur Verarbeitung von SQL-Abfragen gibt. In Java 8 würde man dafür eine externe Bibliothek verwenden, hier käme beispielsweise die Bibliothek *JOOQ* in Frage, wobei die Resultate als Stream zurückgeliefert werden.[MBB06]

Bei LINQ wird zwischen zwei Syntaxen unterschieden. Die eine Syntax wird als *Query-Syntax* bezeichnet, die andere als *Methoden-Syntax*. Die erste ähnelt dabei jener von SQL-Abfragen.[Micb]

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8};

IEnumerable<int> numQuery1 =
    from num in numbers
    where num % 2 == 0
    orderby num
    select num;
```

Da es sich bei LINQ um so genannte erweiterte Methoden handelt, werden den Collections in C-Sharp zusätzlich Methoden zur Verfügung gestellt, welche wie die Streams API Lambda-Ausdrücke im Sinne von Funktionen höherer Ordnung entgegennimmt. Dadurch können Elemente beispielsweise gefiltert oder auf andere Werte gemappt werden. Das bedeutet, dass die obere Query-Syntax vom Compiler in die Methoden-Syntax umgewandelt

wird. Dabei handelt es sich dann um die erweiterten Methoden. Die Methoden-Syntax für obiges Beispiel sieht wie folgt aus.

```
IEnumerable<int> numQuery2 =
    numbers.Where(num => num & 2 == 0)
           .OrderBy(n => n);
```

LINQ unterstützt mehrere Methoden welche in Java 8 noch nicht unterstüzt werden. So wird beispielsweise takeWhile, dropWhile welche erst in Java 9 eingeführt werden ebenfalls bereits von LINQ unterstützt. Im Vergleich zu Java 8 ist bei C-Sharp die Methode *For-Each* nicht direkt in LINQ eingebaut. Im Vergleich zu den parallelen Streams in Java 8 gibt es für LINQ auch ein paralleles Pendant. Dieses wurde mit C-Sharp Version 4.0 hinzugefügt und wird als PLINQ bezeichnet.[Micb]

6.3 Lambdas in C++

In diesem Unterkapitel wollen wir uns kurz die funktionalen Aspekte der Programmiersprache C++ ansehen. Im Bezug auf Lambda-Ausdrücke soll außerdem wie bereits bei C-Sharp ein kurzer Vergleich mit Java durchgeführt werden. Zuerst folgt eine kurze geschichtliche Einleitung hinsichtlich des Hinzufügens von funktionalen Programmierkonzepten zu C++. Lambda-Ausdrücke wurden in C++11 zur Sprache hinzugefügt. Einige Verfeinerungen diesbezüglich wurden anschließend in C++14 veröffentlicht. So kann man seit C++14 unter anderem Default-Werte für Parameter von Lambda-Ausdrücken verwenden. Ebenso können generische Lambda-Ausdrücke über die Variablendeklaration vom Typ *auto* erzeugt werden.[Cpp]

6.3.1 Syntax von Lambda-Ausdrücken

Hier wollen wir uns kurz mit der Syntax von Lambda-Ausdrücken in C++ auseinandersetzen um eventuelle Unterschiede zu Lambda-Ausdrücken in Java herausfiltern zu können.

```
[Capture](Parameter) -> mutable throw() -> Return-Typ {Funktion}
```

Ein Lambda-Ausdruck beginnt in C++ mit eckigen Klammern. Das hat den Vorteil, dass hier nicht zusätzlich ein neues Schlüsselwort zur Sprache hinzugefügt werden musste. Innerhalb der eckigen Klammern können die erfassten Variablen festgelegt werden. Darüber kann der Lambda-Ausdrück auf den umliegenden Kontext zugreifen und somit die Closure bilden. Anschließend daran können in den runden Klammern die Parameter für den Lambda-Ausdruck festgelegt werden. Die Parameterliste ist optional. Danach sehen wir drei weitere optionale Angaben für den Lambda-Ausdruck. Über das Schlüsselwort *mutable* können innerhalb des Lambda-Ausdrucks per Wert erfasste Variablen geändert

werden, da diese standardmäßig als *const* erfasst werden. Mehr dazu im nächsten Unterkapitel über Geltungsbereiche von Variablen innerhalb der Lambda-Ausdrücke in C++. Über *throw()* kann festgelegt werden ob der Lambda-Ausdruck Ausnahmen werfen darf oder nicht. Sofern man den Typ des Rückgabewertes angeben möchte, kann man dies über einen Pfeil gefolgt vom gewünschten Typ angeben. Wie wir sehen werden, kann dieser Typ jedoch vom Compiler hergeleitet werden. In den geschwungenen Klammern folgt der Funktionskörper des Lambda-Ausdrucks. Hier sehen wir einen wesentlichen Unterschied im Vergleich zur Syntax von Lambda-Ausdrücken in Java. Zum einen sind die runden Klammern für die Parameter optional, auch wenn keine Parameter angegeben werden. Dafür müssen hier auf Grund der anderen Syntax immer die geschwungenen Klammern für den Funktionskörper angegeben werden.[Micb]

Als einfaches Beispiel für einen Lambda-Ausdruck nehmen wir wieder die Addition zweier Werte her.

```
auto f = [](int x, int y) { return x+y; };
f(1,2);
```

Dieser Lambda-Ausdruck erfasst keine Variablen vom umliegenden Kontext und nimmt beim Aufruf die beiden Parameter vom Typen *int* entgegen. Die Summe der beiden Werte wird nach Aufruf zurückgegeben. Über das Schlüsselwort *auto* wird der Typ der Variable *f* vom Compiler hergeleitet. Dies hat nicht nur den Vorteil, dass man sich hier keine Gedanken zum Typ des Lambda-Ausdruckes beim Zuweisen zur Variable machen muss. Auch bei den Parametern können wir *auto* verwenden. Dadurch kann man quasi einen generischen Lambda-Ausdruck erzeugen. So könnte man anschließend die Funktion *f* entweder mit Parameter vom Typ *int* aufrufen, aber auch beispielsweise mit Werten vom Typ *double*. Wichtig ist nur, dass die Addition für den jeweiligen Typen festgelegt wird, ansonsten wird der Compiler eine Fehlermeldung ausgeben. So beispielsweise bei der *Addition* zweier Strings. Lambda-Ausdrücke können auch direkt aufgerufen werden, indem man an die Definition des Lambda-Ausdrucks runde Klammern anhängt.

```
auto a = [](int x, int y) {return x+y; }();
```

6.3.2 Erfassung der Variablen

Sehr interessant bei Lambda-Ausdrücken in C++ ist die Umsetzung der Geltungsbereiche der Variablen, beziehungsweise die Erfassung von Variablen vom umliegenden Kontext. Wie wir bei der Vorstellung der Syntax bereits gesehen haben, können jene Variablen, welche für den Lambda-Ausdruck erfasst werden sollen, innerhalb der eckigen Klammern angegeben werden. Hier soll nochmals daran erinnert werden, dass in Java nur effektiv finale Variablen, also jene dessen Werte sich nicht verändert haben und auch innerhalb des Lambda-Ausdrucks nicht verändern, zugelassen sind. Die Erfassung von Variablen soll nachfolgend mit einigen Beispielen erläutert und genauer erklärt werden.

In unserem vorigen Beispiel, bei dem wir die Addition als Lambda-Ausdruck definiert haben, haben wir dem Compiler über die leeren eckigen Klammern mitgeteilt, dass keine Variablen vom umliegenden Kontext für den Lambda-Ausdruck erfasst werden sollen. Daher kann man bei der Angabe von [] nicht auf äußere Variablen innerhalb des Lambda-Ausdrucks zugreifen. Will man hingegen Variablen für die Verwendung erfassen, so gibt das dafür zwei unterschiedliche Möglichkeiten. Variablen können entweder per Referenz oder per Wert erfasst werden. Mit Hilfe des kaufmännischen Und & können Variablen über eine Referenz erfasst werden. Über = lassen sich Variablen über ihren Wert erfassen. Es wird also eine Kopie der Variable, beziehungsweise dessen aktueller Wert bei der Erzeugung des Lambda-Ausdrucks, für den Funktionskörper des Lambda-Ausdrucks erzeugt.[Micb]

Im nachfolgenden Beispiel sollen diese Arten der Variablenerfassung genauer erklärt werden. Dabei sollen über einen Lambda-Ausdruck die Werte zweier Variablen ausgegeben werden.

```
int x = 0;
int y = 0;
//Keine erfassten Variablen => Compilerfehler
auto func = [](){ std::cout << x; };
```

Bei dieser Variante des Beispiels wird der Compiler einer Fehler ausgeben, da in den eckigen Klammern keine Variablen für die Erfassung festgelegt wurden.

```
int x = 0;
int y = 0;
auto func = [=](){ std::cout << x; };

x = 1;
func();
```

Hierbei werden über = Kopien aller im Lambda-Ausdruck referenzierter Variablen angelegt und die Variablen somit über ihren Wert erfasst. Wenn innerhalb des Lambda-Ausdrucks nur auf die Variable x zugegriffen wird, so wird auch nur diese Variable vom Lambda-Ausdruck erfasst. Da dies wie bereits erwähnt zum Zeitpunkt der Erstellung des Lambda-Ausdrucks passiert, wird der Aufruf von *func()* als Wert 0 zurückliefern, obwohl wir davor den Wert von x auf 1 gesetzt haben. Wenn man nur & innerhalb der eckigen Klammern angibt ohne eine Variable dahinter, dann werden standardmäßig alle referenzierten Variablen per Referenz erfasst.

```
int x = 0;
int y = 0;
auto func = [=, &y]() {std::cout << x << " " << y; };

x = 1;
y = 1;
func();
```

Alle referenzierten Variablen außer y werden hier über ihren Wert erfasst, in diesem Fall wird x also nach wie vor über den Wert erfasst. Die Variable *y* hingegen wird über eine Referenz erfasst und daher wird die Änderung der Variable auch innerhalb des Lambda-Ausdrucks sichtbar. Der Aufruf von *func()* liefert für x den Wert 0 und für y den Wert 1 zurück. In C++ 14 wurde des Weiteren die Möglichkeit hinzugefügt, innerhalb der eckigen Klammern neue Variablen für den Lambda-Ausdruck zu initialisieren. Dabei muss kein Typ angegeben werden, da dieser vom Compiler über die Zuweisung hergeleitet werden kann.[Micb]

Wie wir bereits in Java und C-Sharp gesehen haben, gibt es über die Java Streams API respektive LINQ in C-Sharp gute Einsatzmöglichkeiten für anonyme Methoden in der jeweiligen Sprache. In C++ findet man diese Einsatzmöglichkeiten für Lambda-Ausdrücke in der so genannten Standard Template Library, kurz STL.

KAPITEL 7

Laufzeit-Performance

In diesem Kapitel soll die Laufzeit-Performance (nachfolgend als Performance abgekürzt) der Lambda-Ausdrücke mit der jener der alten Konzepte verglichen werden. So soll unter anderem die Ausführungszeit zwischen Lambda-Ausdrücken und anonymen inneren Klassen miteinander verglichen werden. Die Performance der Streams-API soll innerhalb von Java mit der imperativen, vor Java 8 verwendeten Methode zur Verarbeitung größerer Datenmengen verglichen werden. Zusätzlich dazu sollen Performance-Unterschiede zwischen Scala und Java analysiert werden.

Für die Durchführung der Benchmarks wird das OpenJDK Tool JMH verwendet[Orai]. Mit Hilfe dieser Bibliothek ist es relativ einfach möglich Microbenchmarks zu erstellen. Im Bezug auf manuelle Benchmarks gibt es einige Fehlerquellen, welche die Ergebnisse verfälschen können. Gerade durch die Optimierungen, welche durch die JVM über den JIT-Compiler durchgeführt werden, ist es schwer, repräsentative Benchmarks durchzuführen[GLS11]. Aber auch die Garbage-Collection der JVM kann zu Verfälschungen des Benchmarks führen. JMH führt innerhalb eines Benchmarks mehrere Iterationen durch. Innerhalb einer Iteration wird die Methode, welche mit der Annotation *@Benchmark* versehen ist, mehrere Male durchgeführt und deren durchschnittliche Ausführungszeit wird dann je nach Benchmark-Modus in Operationen/Zeiteinheit oder aber in Zeiteinheit/Operation angegeben. Um die JVM und den JIT-Compiler für die Ausführung der zu testenden Methoden aufzuwärmen, werden zuerst Warmup-Iterationen durchgeführt. Diese werden deshalb durchgeführt, damit der JIT-Compiler mögliche Optimierungen durchführen kann[GLS11]. Würde man diese nicht über Warmup-Iterationen ermöglichen, so würde man in einer echten Benchmark-Iteration die Zeit, welche für Optimierungen benötigt wird, ebenfalls messen. Danach folgen die herkömmlichen Iterationen. Über diese wird die Zeit für den Benchmark gemessen. Des Weiteren kann man eine Anzahl von Forks angeben. Diese helfen die Lauf-zu-Lauf Abweichungen der Ausführungszeit der Benchmarks einzuschränken. Für jeden Fork wird eine neue JVM gestartet. Innerhalb dieses Forks werden dann die Warmup- und die normalen Iterationen

des Benchmarks durchgeführt. Wenn man also 10 Iterationen und 5 Forks hat, so werden insgesamt 50 Iterationen durchgeführt.[Orai]

Systeminformationen Nachfolgend sind Informationen für die beiden getesteten JVMs aufgelistet (java -version):

```
Java(TM) SE Runtime Environment
IBM J9 VM (build 2.8, JRE 1.8.0 Windows 8.1
JCL - 20151231_01 based on Oracle jdk8u71-b15
```

```
java version "1.8.0_77"
Java(TM) SE Runtime Environment (build 1.8.0_77-b03)
Java HotSpot(TM) 64-Bit Server VM (build 25.77-b03, mixed mode)
```

Für die durchgeführten Benchmarks wird ein Intel i7-4770k 3,50Ghz mit 16 GB Arbeitsspeicher und einer Samsung 840 EVO SSD verwendet. Als Betriebssystem kommt Windows 10 zum Einsatz. Getestet wird auf zwei verschiedenen JVMs, um zu sehen, welche JVM-Implementierung sich im Bezug auf Lambda-Ausdrücke und Streams in Hinsicht auf Performance durchsetzen kann. So werden die HotSpot-JVM und die J9 von IBM miteinander verglichen.

Bei der HotSpot JVM unterscheidet man zwischen zwei Arten der Kompilierung. Diese werden durch den Modus, in dem die HotSpot-JVM gestartet wird, unterschieden. So gibt es den Client-Modus und den Server-Modus. Im Client-Modus wird ein relativ simpler, dafür aber schnellerer Compiler verwendet. Der Server-Modus ist auf Performance ausgelegt und führt daher eine Reihe von Optimierungen durch[Net07]. Der Server-Modus benötigt im Gegensatz zum Client-Mode der HotSpot-JVM längere Zeit um zu starten. Bei unseren Benchmarks verwenden wir den Server-Modus der HotSpot JVM.

7.1 Lamda-Ausdrücke und anonyme innere Klassen

Hier wollen wir kurz die Performance von Lambda-Ausdrücken und inneren Klassen miteinander vergleichen. Hier implementieren wir das *Callable* Interface und liefern einen Integer zurück.

```
@Benchmark
public Integer callableAnonym() throws Exception{
    Callable<Integer> callable = new Callable<Integer>() {
        @Override
        public Integer call() { return 1; }};
    return callable.call();
}
```

```
@Benchmark
public Integer callableLambda() throws Exception {
    Callable<Integer> callable = () -> 1;
    return callable.call();
}
```

	HotSpot		IBM J9	
Methode	Zeit(ns)	Error+-(ns)	Zeit(ns)	Error+-(ns)
Callable: Anonyme innere Klasse	2,498	0,006	2,371	0,029
Callable: Lambda-Ausdruck	2,626	0,012	5,662	0,117

Tabelle 7.1: Benchmark: Callable Lambda/Anonyme innere Klasse

In Tabelle 7.1 sehen wir, dass die anonyme innere Klasse bei diesem Beispiel etwas schneller als der Lambda-Ausdruck ist. Bei der J9 sind Lambdas um die Hälfte langsamer. Dies könnte daran liegen, dass hier noch nicht dieselben Optimierungen wie bei anonymen inneren Klassen durchgeführt werden. Hier muss man jedoch dazu sagen, dass wir uns im Nanosekundenbereich aufhalten. Als nächstes schauen wir uns deshalb einen Vergleich zwischen Lambda und anonymen inneren Klassen bei Stream-Operationen an.

```
@Benchmark
public long summeLambda() {
    return list.stream()
               .filter(x -> x%2==0)
               .mapToInt(x -> x*2).sum();
}

@Benchmark
public long summeAnonym() {
    return list.stream().filter(new Predicate<Integer>() {
        @Override
        public boolean test(Integer x) {
            return x%2==0;
        }
    }).mapToInt(new ToIntFunction<Integer>() {
        @Override
        public int applyAsInt(Integer x) {
            return x*2;
        }
    }).sum();
}
```

Anzahl	Methode	HotSpot		IBM J9	
		Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
100.000	anonyme innere Klasse	0,148	0,005	0,278	0,001
	Lambda	0,148	0,004	0,172	0,010
1.000.000	anonyme innere Klasse	4,162	0,021	2,932	0,022
	Lambda	4,065	0,013	1,864	0,026

Tabelle 7.2: Benchmark: Streams Lambda/anonyme innere Klasse

In 7.2 sehen wir, dass bei der HotSpot JVM anonyme innere Klassen und Lambdas in etwa gleich schnell sind. Bei der J9 ist die Umsetzung mit Lambda-Ausdrücken jeweils um etwa ein Drittel schneller als mit anonymen inneren Klassen. Der J9 JIT Compiler kann hier bei Lambda-Ausdrücken bessere Optimierungen des JVM-Bytecodes in Maschinencode vornehmen. Bei mehreren Elementen sind die Lambda-Ausdrücke etwas schneller als anonyme innere Klassen.

7.2 Stream-Performance

In diesem Unterkapitel liegt das Hauptaugenmerk auf der Performance von Streams innerhalb von Java 8. So soll unter anderem die Performance von parallelen Streams im Vergleich zu sequenziell ausgeführten Streams verglichen werden. Zusätzlich dazu soll auch zwischen der Streams API und der vor Java 8 verwendeten Variante, nämlich der Verarbeitung von Datenmengen in *Collections* über ein oder mehrere *for*-Schleifen verglichen werden. Außerdem wollen wir hier feststellen, wann es sich von der Performance her auszahlt, parallele Streams zu verwenden und wann nicht. Am Ende werden wir noch einige Fallen in Hinsicht auf die Performance von parallelen Streams präsentieren.

Betrachten wir zunächst die Ergebnisse eines relativ simplen Benchmarks. Hier sollen jeweils die Zahlen einer Array-Liste, über eine *for*-Schleife, einen sequenziellen und einen parallelen Stream addiert werden. Es werden dazu über das Benchmark-Tool JMH jeweils 10 Warmup-Iterationen gefolgt von 10 Iterationen mit 5 Forks durchgeführt. Die Anzahl der Elemente beträgt 1 Million. Die Zeit bezieht sich auf Millisekunden pro Operation. Nachfolgend der Aufbau des Benchmarks mit JMH.

```
@Setup
public void setup() {
    list = new ArrayList<>();
    for(int i=1; i<=1000000; i++) {
        list.add(i);
    }
}
```

Über *setup* wird die Array-Liste mit einer Million Einträge erzeugt. Dies geschieht immer am Anfang einer Iteration und die Erstellung der Liste wird nicht in die Ausführungszeiten miteinbezogen.

```
@Benchmark
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@BenchmarkMode(Mode.AverageTime)
public int forloop() {
    int x = 0;
    for(Integer i : list) {
        x = x+i;
    }
    return x;
}
```

Hier der Aufbau des Benchmarks für die for-Schleife. Über *@Benchmark* wird die Methode als zu testende Methode festgelegt. Über *@OutputTimeUnit* wird die Zeiteinheit für die Messungen angegeben. Die Annotation *BenchmarkMode* legt hier fest, das die Durchschnittszeit, welche für die Ausführung der Methode benötigt wird, ausgegeben werden soll. Innerhalb der Methode wird über eine for-Schleife über die *for-Each*-Syntax jedes Element der Liste über einen Iterator durchlaufen und zur Summe x dazugezählt. Das Ergebnis wird zurückgeliefert, da der JIT-Compiler die Berechnung ansonsten *wegoptimieren* würde. Sofern das Ergebnis nicht weiterverwendet wird, wird der Code vom JIT-Compiler als *Dead-Code*, also Code dessen Ergebnis nicht weiterverwendet wird, erkannt und dem entsprechend nicht mehr ausgeführt. Anschließend sehen wir den Code für den sequenziellen Stream. Die Annotationen sind nachfolgend bei allen Methoden gleich.

```
public int sequentialStream() {
    return list.stream()
               .mapToInt(Integer::intValue)
               .reduce(0,(x,y) -> x+y);
}
```

Von der erzeugten Liste der zu addierenden Zahlen wird ein Stream angefordert. Anschließend wird ein Mapping durchgeführt wobei die Integer-Elemente in Elemente vom primitiven Typ *int* umgewandelt werden. Daher liefert *mapToInt* einen *IntStream* zurück. Anschließend wird über *reduce* die Summe berechnet. In diesem Benchmark wird der sequenzielle Stream auf drei verschiedene Arten durchgeführt. Eine Variante ist der obere Code ohne die Verwendung von *mapToInt(Integer::intValue)*. Das Problem dabei ist Boxing und Unboxing der Elemente der Liste vom Typ Integer zu *int*. Wie wir bei den Ergebnissen des Benchmarks sehen werden, erhöht dies den Rechenaufwand. Bei *reduce*

gibt es mehrere Möglichkeiten. So könnte man auf einen *IntStream* für die Berechnung der Summe ebenfalls die Methode *sum* auf den Stream anwenden. Betrachten wir nun die Ergebnisse des Benchmarks in 7.3. Hier werden mehrere Vergleiche auf einmal dargestellt.

	HotSpot		IBM J9	
Methode	Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
iterator	1,167	0,020	1,509	0,011
spliterator forEachRemaining	1,283	0,024	1,274	0,005
spliterator tryAdvance	2,367	0,058	4,169	0,025
for-Schleife	1,206	0,017	1,667	0,326
for-Each-Schleife	1,218	0,008	1,560	0,016
sequenzieller Stream	1,218	0,030	1,754	0,426
paralleler Stream	1,218	0,030	0,887	0,022
seq. ohne MapToInt	4,246	0,043	5,461	0,063
par. ohne MapToInt	2,647	0,023	2,920	0,022
seq. mit sum	5,000	0,020	1,336	0,011
par. mit sum	1,214	0,019	0,912	0,046

Tabelle 7.3: Benchmark: Addition von Zahlen, Anzahl der Elemente N = 1000000

In den ersten drei Zeilen haben wir einen Vergleich zwischen Iteratoren und Spliteratoren. Bei Spliteratoren haben wir einmal die Umsetzung mittels *forEachRemaining* und einmal über *tryAdvance*. Die Methode *tryAdvance* vereint hier *hasNext* und *next* des Iterators. Bei der J9 ist der Spliterator schneller als der Iterator. Bei der Hotspot JVM sind Iteratoren etwas schneller als Spliterator. Der manuell verwendete Spliterator mit der Methode *tryAdvance* schneidet hier langsamer ab als *forEachRemaining*. Das liegt daran, dass *forEachRemaining* mit einer *while* Schleife direkt über alle Elemente iteriert. Die Methode *tryAdvance* hingegen wendet die Funktion nur auf das nächste Element an und liefert anschließend einen boolean zurück. Dadurch muss *tryAdvance* wie beim Iterator mehrere Methodenaufrufe durchführen.

Die *for*-Schleife ist im Vergleich zum sequenziellen Stream in etwa gleich schnell auf Seiten der HotSpot-JVM. Der parallele Stream ist bei HotSpot mit N=1000000 etwa gleich schnell wie die sequenzielle Variante. Das liegt am zusätzlichen Aufwand, um die parallele Verarbeitung zu ermöglichen. Bei der J9 von IBM läuft die parallele Variante schneller ab. Der parallele Stream auf der J9 mit *mapToInt*, um Boxing und Unboxing bei der Addition zu umgehen, bringt hier die besten Ergebnisse. Dafür sind die Ergebnisse für sequenzielle Streams und die herkömmliche Schleife bei J9 langsamer als auf der HotSpot-JVM. Ungewöhnlich ist die hohe Abweichung für den sequenziellen Stream bei der J9. Hier wird während einer Iteration vermutlich die *Garbage Collection* ausgelöst. Dadurch kommt die hohe Abweichung zwischen den einzelnen Forks zustande.

Bei dem sequenziellen Stream, wo statt *reduce* die Methode *sum* verwendet wird, gibt es auf Seiten der HotSpot-JVM ein Problem. Nach einigen Iterationen steigt die Ausführungszeit auf knapp das Dreifache an. Auf der J9 hingegen läuft diese Variante im Vergleich zum

sequenziellen Stream mit *reduce* gleichmäßiger ab. Bei der parallelen Berechnung der Summe über *sum* treten diese Probleme bei der HotSpot-JVM nicht auf. Die beiden Varianten ohne *mapToInt* zeigen, wie wichtig es im Hinblick auf den Einsatz von Streams ist, sich mit Boxing und Unboxing von Elementen zu beschäftigen. Es wird für die Umwandlung von Elementen des Typs *Integer* zum primitiven Typ *int* und umgekehrt relativ viel zusätzliche Zeit benötigt. Verwendet man also nicht den spezialisierten Stream für primitive Datentypen oder führt explizit kein Unboxing über *mapToInt* für alle Elemente durch, kann es sehr schnell passieren, dass die Anwendung von Streams im Vergleich zur *for*-Schleife langsamer abläuft.

7.2.1 Performance der Operationen auf Streams

In diesem Unterkapitel wollen wir die Performance-Unterschiede einzelner Zwischenoperationen und terminierenden Operationen in Hinblick auf die Umsetzung mit Schleifen, sequenziellen und parallelen Streams genauer betrachten. Auf Grund dieser Benchmarks können wir den Overhead, der durch Streams erzeugt wird, besser nachvollziehen.

filter Aus einer Liste mit zehn Millionen Zahlen wollen wir die geraden Zahlen herausfiltern. Die Zahlen sind fortlaufend gewählt, dadurch werden genau die Hälfte der Elemente herausgefiltert und die restlichen Zahlen werden zurückgeliefert. Bei einer Variante des Beispiels werden wir auch Methodenreferenzen verwenden. Der Berechnungsaufwand pro Element ist hier relativ gering. Count wird als terminierende Operation gewählt, damit der Stream durchlaufen wird.

```
Stream: (sequenziell / parallel / Methodenreferenz)
integerList.stream().filter(x -> x % 2 = 0).count()

Schleife:
for (Integer integer : integerList) {
    if(integer % 2 == 0) {
        count++;
    }
}
```

In Tabelle 7.4 sehen wir, dass die J9 JVM etwas schneller als die Hotspot JVM ist. Die Umsetzungen über die Schleife und den sequenziellen Stream sind hier auf beiden JVMs in etwa gleich schnell. Hier fällt auf, dass bei J9 der sequenzielle Stream schneller als die Schleife ist, jedoch ist hier die Abweichung im Bereich von 1,5 ms, daher kann man hier nicht eindeutig sagen, dass sequenzielle Streams schneller sind. Zwischen Streams mit Lambda-Ausdrücken und Methoden-Referenzen gibt es zeitlich gesehen auch kaum Unterschiede. Der parallele Stream kann wie erwartet besser abschneiden als der sequenzielle. Jedoch haben wir hier eine knappe Performance-Steigerung von einhundert Prozent, obwohl unser Testsystem acht Kerne besitzt. Auf Grund dessen,

	HotSpot		IBM J9	
Methode	Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
filter schleife	22,136	0,312	21,066	0,315
filter sequenziell	23,829	0,277	20,564	1,564
filter sequenziell method-reference	23,923	0,467	19,928	0,904
filter parallel	12,107	0,582	10,857	1,143
filter parallel method-reference	12,355	0,052	10,630	1,038

Tabelle 7.4: Benchmark: Filter, Anzahl der Elemente 10 Millionen

dass die Anzahl der Tasks und auch die Teilbarkeit durch die Verwendung von Listen für parallele Verarbeitung von Vorteil sind, jedoch der Aufwand für das Filtern der einzelnen Elemente sehr gering ist, kann hier keine achtfache Verbesserung der Performance erreicht werden.

flatMap Im nächsten Benchmark in Tabelle 7.5 schauen wir uns die Performance der Stream-Operation *flatMap* im Vergleich zu verschachtelten Schleifen an. Dabei erzeugen wir eine neue Klasse, die lediglich eine Liste von Werten vom Typ Integer beinhaltet. Die Liste enthält im folgenden Benchmark einhunderttausend Elemente. Diese wiederum haben jeweils eine Liste mit einhundert Elementen vom Typ Integer. Insgesamt verarbeiten wir hier zehn Millionen Integer-Werte. Wie wir in Tabelle 7.5 sehen können, ist die frühere Variante mit verschachtelten Schleifen schneller als sequenzielle Streams. Das liegt daran, dass wir bei *flatMap* einen höheren Overhead haben als bei den Schleifen. Die Hotspot JVM ist in diesem Benchmark schneller als die IBM J9.

```
@Benchmark
public Integer flatMapStream() {
  return elementList.stream()
           .flatMap(element -> element.getList().stream())
           .reduce(0, (x,y) -> x+y);
}

@Benchmark
public Integer flatMapLoop() {
  Integer sum = 0;
  for(Element element : elementList) {
    for(Integer x : element.getList()) {
      sum += x;
    }
  }
  return sum;
}
```

	HotSpot		IBM J9	
Methode	Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
flatMap Schleife	33,941	0,629	51,932	0,124
flatMap sequenzieller Stream	48,128	2,579	79,180	1,727
flatMap paralleler Stream	19,376	0,138	30,149	3,353

Tabelle 7.5: Benchmark: FlatMap Schleife/Streams

grouping Um die Operation *grouping* zu testen, verwenden wir ein ähnliches Beispiel wie aus Unterkapitel 4.8 *Vergleich zwischen Streams und Schleifen*. Nachfolgend die Variante mit und anschließend ohne Verwendung von Streams.

```
@Benchmark
public Map<String, Map<String, List<Student>>> groupSeq() {
  return studenten.stream()
    .collect(Collectors.groupingBy(Student::getGeschlecht,
         Collectors.groupingBy(Student::getStaatsbuergerschaft)));
}

@Benchmark
public Map<String, Map<String, List<Student>>> groupLoop() {
  Map<String, Map<String, List<Student>>> multimap =
      new HashMap<>();

  for (Student student : studenten) {
    if (multimap.get(student.getGeschlecht()) == null) {
      multimap.put(student.getGeschlecht(), new HashMap<>());
    }
    Map<String, List<Student>> map =
        multimap.get(student.getGeschlecht());
    if (map.get(student.getStaatsbuergerschaft()) == null) {
      map.put(student.getStaatsbuergerschaft(),
              new ArrayList<>());
    }
    List<Student> studenten =
        map.get(student.getStaatsbuergerschaft());
    studenten.add(student);
  }
  return multimap;
}
```

Hier wollen wir wieder die Performance zwischen Streams und Schleifen testen. Dazu werden wir eine Million Studenten als Liste verwenden. Fünfzig Prozent der Studenten

sind männlich, fünfzig Prozent weiblich. Die Staatsbürgerschaft werden wir auf zehn Länder aufteilen, wobei jedes Land zehn Prozent der Studenten gleichmäßig zugewiesen bekommt. In Tabelle 7.6 sehen wir das Ergebnis des Grouping-Benchmarks. Hier sehen wir, dass die Schleife etwas langsamer als der Stream abläuft. Grund dafür könnte die von uns gewählte Implementierung sein, die man effizienter umsetzen könnte. Die Umsetzung der Operation innerhalb der Streams API ist hier effizienter als unsere Implementierung.

	HotSpot		IBM J9	
Methode	Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
grouping Schleife	30,878	1,022	33,847	0,366
grouping sequenzieller Stream	28,116	0,839	40,605	0,741
grouping paralleler Stream	12,394	0,606	12,672	0,283

Tabelle 7.6: Benchmark: Grouping Schleife/Streams

7.2.2 Parallele Streams

In diesem Unterkapitel soll geklärt werden, wann sich der Einsatz von parallelen Streams lohnt und wann man besser bei sequenziellen Streams bleiben sollte. Grundlage für dieses Unterkapitel bietet zunächst ein Paper von Doug Lea mit dem Titel *When to use parallel streams*[L+14]. Anschließend werden mit JMH weitere Microbenchmarks im Bezug zu sequenziellen und parallelen Streams durchgeführt.

Im Zusammenhang mit parallelen Streams kommt es immer darauf an, über welche Art von *Collection* der jeweilige Stream erzeugt wird. Wenn als Quelle für den Stream eine IO-basierte Quelle verwendet wird, beispielsweise beim Einlesen einer Datei, so sollte ein sequenzieller Stream verwendet werden. Gut geeignet für parallele Streams sind ArrayListen, Arrays und HashMaps. Weniger gut geeignet sind dagegen verlinkte Listen. Dabei kommt es darauf an, wie gut die Collections für die parallele Verarbeitung in kleinere Teile aufgeteilt werden können und ob während des Zugriffs auf die Elemente ein Synchronisationsmechanismus zum Einsatz kommt.[L+14]

Ein weiterer sehr wichtiger Faktor bei der Überlegung, ob man einen Stream sequenziell oder parallel ausführen sollte, ist der Berechnungsaufwand, der zwischen den einzelnen Threads aufgeteilt werden kann. Dabei spielt auch die Anzahl der zu verarbeitenden Elemente eine Rolle. In [L+14] wird eine kurze Formel diesbezüglich vorgestellt. Wenn man von einer Operation F und von einer Gesamtanzahl von Elementen N aus. Die Ausführung der Operation F auf ein Element hat die Kosten Q. Die folgende Formel sollte erfüllt sein um einen parallelen Stream effizient verwenden zu können.

```
N * Q >= 10000
```

Sofern man sicher gehen möchte, sollte man noch ein bis zwei Nullen anhängen, also N*Q sollte mindestens einhunderttausend beziehungsweise eine Million betragen, um bei

einer parallelen Ausführung keinen Verlust der Performance auf Grund des zusätzlichen Aufwands beim Erstellen und der Aufteilung der Teilaufgaben zu erhalten. Die Kosten für Q können dabei über die Anzahl der Operationen geschätzt werden. Führt man beispielsweise ein Mapping von einer Zahl x auf x+1 durch, so kann Q = 1 angenommen werden, da bei jedem Element lediglich zwei Zahlen addiert werden. Je aufwendiger die durchzuführenden Operationen sind, desto früher – in Hinsicht auf die Anzahl der Elemente – lohnt sich der Einsatz von parallelen Streams.[L+14]

Um für das vorherige Beispiel 7.3 abschätzen zu können, ab welcher Größe der Liste man sich die parallele Verarbeitung des Streams überlegen sollte, folgen für die ersten drei Methoden weitere Versuche mit unterschiedlicher Größe der Liste. Für einen besseren Vergleich sind die jeweiligen Ergebnisse für N = 1000000 ebenfalls in Tabelle 7.7. Diese zeigt die Ergebnisse für die Umsetzung mit *for*-Schleife, sequenziellem und parallelem Stream für Listen unterschiedlicher Größe.

Anzahl	Methode	HotSpot		IBM J9	
		Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
100	for-Schleife	0,079 μs	0,001 μs	0,130 μs	0,001 μs
	sequenzieller Stream	0,140 μs	0,006 μs	0,148 μs	0,004 μs
	paralleler Stream	7,868 μs	0,115 μs	13,957 μs	0,418 μs
100.000	for-Schleife	0,069	0,001	0,129	0,033
	sequenzieller Stream	0,077	0,001	0,077	0,001
	paralleler Stream	0,110	0,002	0,032	0,001
500.000	for-Schleife	0,497	0,005	0,815	0,163
	sequenzieller Stream	0,493	0,005	0,528	0,006
	paralleler Stream	0,534	0,007	0,276	0,049
1.000.000	for-Schleife	1,218	0,008	1,560	0,016
	sequenzieller Stream	1,218	0,030	1,754	0,426
	paralleler Stream	1,218	0,030	0,887	0,022
2.500.000	for-Schleife	3,300	0,052	4,076	0,504
	sequenzieller Stream	3,250	0,024	3,264	0,034
	paralleler Stream	2,782	0,215	2,424	0,068
5.000.000	for-Schleife	6,949	0,057	7,921	0,851
	sequenzieller Stream	6,943	0,046	6,491	0,060
	paralleler Stream	5,548	0,063	4,716	0,035
10.000.000	for-Schleife	15,245	0,122	14,743	0,140
	sequenzieller Stream	15,072	0,112	12,764	0,167
	paralleler Stream	11,093	0,259	9,339	0,057

Tabelle 7.7: Benchmark: Addition von Zahlen mit unterschiedlich großen Listen

Bei einer Größe von einhundert Elementen, kann man den Overhead der parallelen Streams gut erkennen, auch wenn wir uns hier im Mikrosekunden-Bereich aufhalten. Man sieht hier auch die Kosten, welche für die Erzeugung des sequenziellen Streams im

Vergleich zur Schleife anfallen. In Tabelle 7.7 erkennt man, dass man bei der HotSpot-JVM für die parallele Ausführung für dieses Beispiel in etwa eine Million Einträge innerhalb der Liste benötigt, um einen Performance-Vorteil im Vergleich zur sequenziellen Ausführung zu erhalten. N * Q beträgt in diesem Fall eine Million. Wobei Q, also der Aufwand für den einzelnen Task, auf Grund der Addition auf 1 geschätzt wird. Auf Seiten der J9-JVM scheint dieser Wert wesentlich geringer auszufallen, da bereits bei einer Anzahl von einhunderttausend die parallele Verarbeitung mehr als doppelt so schnell als die sequenzielle Berechnung ausfällt.

7.2.3 Performanceunterschiede bei verschiedenen Quellen

Wie bereits erwähnt, hängt die Performance von parallelen Streams auch von der Quelle des Streams ab, da unterschiedliche Collections unterschiedliche Zugriffszeiten auf einzelne Elemente haben. Das wollen wir in diesem Unterkapitel näher betrachten. Dazu verwenden wir wieder eine terminierende Operation, um die Abarbeitung des Streams anzustoßen. Um den Aufwand möglichst klein zu halten wird *count* verwendet. Die Tests sind dabei so aufgebaut, dass die jeweiligen Stream-Quellen mit einer Million Elementen befüllt sind. Getestet werden Arrays, Array-Listen, verlinkte Listen, HashSets, die synchronisierte Methode *Math.random* und *Files.lines*.

	HotSpot		IBM J9	
Methode	Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
array sequenziell	4,191	0,063	0,399	0,002
array parallel	0,785	0,004	0,068	0,001
liste sequenziell	5,794	0,027	0,400	0,001
liste parallel	0,831	0,007	0,068	0,001
set sequenziell	15,886	0,212	8,845	0,404
set parallel	6,394	0,912	3,956	0,214
io sequenziell	25,411	0,165	22,024	0,133
io parallel	36,267	0,777	23,928	0,490
verlinkt sequenziell	9,286	0,070	6,027	0,727
verlinkt parallel	27,267	0,253	11,617	6,303
math.random sequenziell	19,397	0,086	25,405	8,229
math.random parallel	226,457	5,761	230,037	8,162

Tabelle 7.8: Benchmark: Stream-Quellen, Anzahl der Elemente 1 Million

Bei Tabelle 7.8 sehen wir, dass man bei Arrays, bei Array-Listen und bei HashSets mit Hilfe von parallelen Streams eine gute Performance-Steigerung erzielen kann. Bei dem Test mit *IO* haben wir eine Textdatei über *Files.lines* eingelesen. Dabei entspricht jede Zeile einem Element des erzeugten Streams. Hier wurde eine Datei mit einer Million Zeilen verwendet. Hier sieht man, dass sich parallele Streams nicht bei jeder Stream-Quelle eignen. Die parallele Umsetzung ist hier sogar langsamer als die sequenzielle.

Bei verlinkten Listen ist die sequenzielle Variante dreimal so schnell. Bei dem über *Math.random* erzeugten Stream ist die parallele Variante mehr als 10 Mal langsamer. Grund dafür ist, dass *Math.random* eine synchronisierte Methode ist. Beim Vergleich der JVMs ist fast überall die J9 schneller als die Hotspot JVM. Vorallem bei den Arrays und Listen ist die J9 wesentlich schneller. Das liegt daran, dass hier Code vom J9 JIT-Compiler wegoptimiert wird.

7.2.4 Collect und Reduce

Auch bei den *Collect- und Reduce-Operationen* kommt es darauf an, in welche Collection die Elemente des Streams zusammengefügt werden. Wenn wir beispielsweise *reduce* verwenden um die Gesamtlänge der Strings herauszufinden, so müssen bei einer parallelen Ausführung lediglich die Teilsummen addiert werden. Dafür benötigen wir so gut wie keine Rechenleistung. Wenn wir aber über *collect* wie im nachfolgenden Benchmark Listen zusammenfügen, so müssen bei der parallelen Ausführung die Teillisten zusammengefügt werden. Dies kann je nach verwendeter Collection einen relativen hohen Aufwand bedeuten.

Als Quelle für den Stream verwenden wir das Array vom vorherigen Beispiel, welches mit einer Million Strings befüllt wurde. Hier testen wir die Performance der Hilfsklasse *Collectors*. Dabei werden wir die Methoden *toList*, *toSet*, *toMap* und *toConcurrentMap* und *joining* miteinander vergleichen. Die Methode *joining* wird dabei ebenfalls mit einer for-Schleife mittels *StringBuilder* umgesetzt.

	HotSpot		IBM J9	
Methode	Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
collect toList for	4,894	0,177	4,081	0,136
collect toList seq	5,822	0,177	5,187	0,753
collect toList par	5,625	0,052	6,764	0,036
collect toSet seq	33,838	0,468	45,559	2,871
collect toSet par	90,481	3,216	285,212	19,140
collect toMap seq	35,956	0,730	50,513	1,561
collect toMap par	88,419	3,871	287,599	35,181
collect concurrent seq	83,968	1,966	118,784	1,606
collect concurrent par	31,833	4,024	40,879	1,918
collect joining for	29,693	1,791	27,951	0,470
collect joining seq	29,268	0,405	28,380	0,319
collect joining par	40,129	0,380	41,161	0,458

Tabelle 7.9: Benchmark: collect, Stream-Quelle: Array mit 1 Million Strings

In der Tabelle 7.9 sehen wir die Ergebnisse des Collect-Benchmarks. Das Sammeln der Werte in eine Liste ist über eine Schleife schneller als über Streams. Beim Zusammenfügen von Listen haben wir kaum einen Overhead beim parallelen Stream, daher ist die

Ausführungszeit zwischen sequenziellem und parallelem Stream in etwa gleich. Man darf hier nicht vergessen, dass wir keine Zwischenoperationen verwenden und lediglich den Overhead der parallelen Streams betrachten. Das Zusammenfügen von Sets beansprucht schon mehr Zeit. Dadurch ist die Ausführungszeit beim parallelen Stream wesentlich langsamer. In den nächsten vier Zeilen sehen wir den Vergleich zwischen Maps und ConcurrentMaps. Beim Collector im Zusammenhang mit *toMap* erzeugt jeder Task seine eigene Hashmap. Die Ergebnisse werden dann durch Zusammenfügen der Hashmaps erzeugt. Bei *ConcurrentMaps* wird lediglich eine Map erzeugt. Auf diese kann auf Grund der Thread-Sicherheit von allen Tasks aus zugegriffen werden. Hier spart man sich den Schritt des Zusammenfügens der Hashmaps[Orab]. Dadurch sind ConcurrentMaps wesentlich besser für parallele Streams geeignet. Daher schlägt die parallele ConcurrentMap-Variante sogar die sequenzielle toMap-Methode. Zwischen den JVMs ist hier die Hotspot JVM performancetechnisch vorne. Bei der J9 besitzen die parallelen collect-Operationen einen wesentlich höheren Overhead als bei der Hotspot JVM.

Wenn wir *joining* über eine Schleife mit *+=* zum Resultat-String hinzufügen, so erzeugen wir bei jedem Testlauf eine Million Strings, da Strings *immutable* sind. Stattdessen haben wir einen StringBuilder verwendet, um dieses Problem zu umgehen. Hier sehen wir wieder den Vorteil von Streams, da wir uns keine Gedanken über die darunterliegende Implementierung machen müssen. Hier ist die parallele Variante auf Grund der Immutability von Strings langsamer. Die einzelnen Threads müssen die Ergebnisse zusammenfügen, dabei müssen jedes Mal neue Strings erzeugt werden.

7.2.5 Größe und Anzahl der Tasks

Die Anzahl der zu verarbeitenden Elemente beeinflusst wie viele Elemente jeweils pro Task abgearbeitet werden. Die Anzahl der erzeugten Tasks ist im Code vorgegeben. Diese berechnet sich derzeit mit folgender Formel: *(Anzahl der Kerne - 1) * 4*. In unserem Fall werden also achtundzwanzig Tasks für die parallele Verarbeitung erzeugt. Wie wir bereits im ersten Benchmark gesehen haben, spielt auch die Komplexität der Tasks für die parallele Ausführung eine Rolle. So hatten wir bei der Addition von Zahlen eine sehr geringe Komplexität. Daher konnten auch keine so hohen Performance-Vorteile erzielt werden. Der zusätzliche Aufwand für die Organisation der parallelen Abarbeitung und anschließendem Zusammenfügen der Ergebnisse war in etwa gleich groß, wie die Performance-Verbesserung der parallelen Ausführung. Im nachfolgenden Benchmark wollen wir betrachten, wie sich die Performance bei komplexeren Tasks verhält. Dazu verwenden wir die Methode *Blackhole.consumeCPU(long tokens)* von JMH. Diese kann CPU-Berechnungen simulieren. Dazu übergibt man der Methode die Anzahl an Tokens die berechnet werden soll. Komplexere Berechnungen simulieren wir mit höherer Anzahl an Token. Nachfolgend rufen wir die Methode mit zehn Millionen Token auf. Ziel dieses Benchmarks soll es sein, aufzuzeigen, dass bei sehr komplexen Tasks die Anzahl der Elemente für eine parallele Verbesserung der Performance weniger relevant wird.

In Tabelle 7.10 sehen wir in der linken Spalte die Anzahl der Elemente. Die Methode *consume* wird hier in eine filter-Operation eingebaut, bevor der Wert *true* zurückgegeben

	HotSpot				IBM J9			
	sequenziell		parallel		sequenziell		parallel	
N	Zeit(ms)	+-(ms)	Zeit	+-	Zeit	+-	Zeit	+-
1	18,666	0,097	18,683	0,079	22,368	0,151	22,404	0,099
2	37,290	0,170	18,858	0,170	44,891	0,508	22,574	0,157
3	55,901	0,262	19,125	0,262	67,166	4,022	22,932	0,104
4	74,665	0,386	19,438	0,386	90,408	3,376	23,301	0,110
5	93,636	0,434	20,469	0,434	116,063	4,894	23,753	0,022
6	111,705	0,744	20,618	0,744	134,688	11,327	23,765	0,015
7	130,600	0,880	20,533	0,880	161,565	0,684	23,865	0,088
8	149,550	0,751	20,945	0,751	180,936	8,791	24,604	1,439
16	298,522	2,077	42,166	2,077	368,658	2,226	49,730	3,519
32	596,341	2,768	83,280	2,768	706,846	26,889	97,811	5,013
100	1870,509	28,018	275,35	27,018	2311,249	34,076	331,086	30,341

Tabelle 7.10: Benchmark: Task-Komplexität, N=Anzahl der Elemente

wird. Wir simulieren hier also eine komplexe filter-Operation. Man sieht hier sehr schön, wie sich die parallele Ausführung bereits ab zwei Elementen auszahlt. Da wir eine CPU mit acht Kernen verwenden, bleibt die Ausführungszeit bis zu einer Anzahl von acht Elementen hier relativ ähnlich. Hier kann die Parallelität optimal ausgenutzt werden. Dies verhält sich bei der IBM J9 genauso. Jedoch benötigen wir hier ein wenig mehr Zeit für die Ausführung im Gegensatz zur HotSpot JVM.

7.2.6 Vergleich zwischen parallelen Streams und ForkJoin

Nun werden wir die Performance von parallelen Streams direkt mit dem Fork-Join-Framework vergleichen. Dazu verwenden wir das Beispiel aus Unterkapitel 4.8.2 *Parallele Verarbeitung der Daten*. Zusätzlich zeigen wir bei diesem Benchmark auch die sequenzielle Performance als Vergleich. Nachfolgend sehen wir die getesteten Methoden.

```
public List<Statistik> stream() {
    return studenten.stream()
                    .map(Student::berechneStatistik)
                    .collect(toList());
}

public List<Statistik> streamPar() {
    return studenten.parallelStream()
                    .map(Student::berechneStatistik)
                    .collect(toList());
}
```

```
public List<Statistik> schleife() {
    List<Statistik> statistiken = new ArrayList<>();
    for(Student student : studenten) {
        statistiken.add(student.berechneStatistik());
    }
    return statistiken;
}

public List<Statistik> forkJoin() {
    StatistikRecursiveTask task =
        new StatistikRecursiveTask(studenten);
    return task.invoke();
}
```

Die Methode *berechneStatistik* nimmt dabei eine Liste von Zeugnissen des Studenten und berechnet den Notendurchschnitt gewichtet nach ECTS. Bei der Variante mit dem *RecursiveTask* können wir die Grenze angeben, ab welcher die Berechnung sequenziell ausgeführt werden soll und nicht weiter aufgeteilt wird. Diese Grenze ist beim nachfolgenden Benchmark auf *Anzahl der Elemente dividiert durch 28* gesetzt. Dadurch werden also gesamt 28 Tasks für die parallele Verarbeitung erzeugt. Das ist jener Wert, der auch bei parallelen Streams auf Grund der 8 Kerne der CPU verwendet wird. Danach werden wir unterschiedliche Grenzen auf Performance Unterschiede testen.

Anzahl	Methode	HotSpot		IBM J9	
		Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
100.000	Schleife	70,311	1,124	64,153	0,580
	sequenzieller Stream	70,584	1,088	62,950	0,289
	Fork Join	33,701	0,193	29,025	0,643
	paralleler Stream	32,716	0,601	27,759	0,482
200.000	Schleife	136,925	1,204	130,793	6,718
	sequenzieller Stream	137,717	1,396	132,608	2,587
	Fork Join	67,261	0,580	56,032	1,106
	paralleler Stream	65,642	0,287	56,817	0,517

Tabelle 7.11: Benchmark: Stream und Fork Join, pro Student 100 Zeugnisse

Der sequenzielle Stream ist in 7.11 wieder etwas langsamer als die Schleife. Der parallele Stream läuft hingegen schneller ab als die Lösung mittels Fork Join Framework. Auch wenn die Unterschiede hier minimal sind. Die *IBM J9* ist hier performanter als die *HotSpot JVM*.

Betrachten wir nun die Ergebnisse unterschiedlicher Grenzen des *RecursiveTasks* in Tabelle 7.12. Bei einer Grenze von eins müssen sehr viele Tasks erzeugt werden. In unserem Beispiel sind es einhunderttausend. Dadurch entsteht ein höherer Overhead

	HotSpot		IBM J9	
Grenze	Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
1	40,532	2,631	32,706	0,530
10	34,542	0,251	29,495	0,150
100	34,091	0,245	28,454	0,028
1.000	33,366	0,164	28,251	0,272
3.571	33,701	0,193	28,826	0,435
10.000	34,362	0,239	31,474	4,206
50.000	44,292	1,015	37,527	0,659
100.000	70,648	0,749	92,962	2,290

Tabelle 7.12: Benchmark: Unterschiedliche Grenzen bei RecursiveTask. Studenten: 100.000, Zeugnisse jeweils 100

für die Vorbereitung der parallelen Verarbeitung. Ab einer zu hohen Grenze werden sehr wenige Tasks erzeugt, beziehungsweise eine Anzahl an Tasks, welche sich schlecht auf unsere acht Kerne aufteilen lässt. So haben wir bei einer Grenze von zehntausend genau zehn erzeugte Tasks. Wenn alle Threads in etwa gleich lange für die Verarbeitung brauchen, so können am Ende nur mehr zwei Threads an den übrigen Tasks weiterarbeiten. Die restlichen Threads haben nichts mehr zu tun. Die Grenze mit einhunderttausend erzeugt nur mehr einen Task. Daher läuft es in etwa so schnell ab, wie die sequenzielle Verarbeitung in unserem vorherigen Benchmark. Bei der IBM J9 läuft es hier sogar wesentlich langsamer als bei der sequenziellen Variante.

7.2.7 Fallstricke paralleler Streams

Hier wollen wir uns noch ein paar Spezialfälle im Zusammenhang mit parallelen Streams ansehen, die zu Performance-Einbußen führen können. Dazu schauen wir uns zunächst die Benchmarks für die Zwischenoperation *distinct* an.

	HotSpot		IBM J9	
Methode	Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
distinct seq	37,709	1,541	28,261	1,434
distinct par	61,724	9,541	63,825	0,799
distinct unordered seq	37,160	1,583	27,142	2,430
distinct unordered par	21,687	0,253	22,873	0,406

Tabelle 7.13: Benchmark: distinct, Stream-Quelle: Liste mit 1 Million Strings

Bei *distinct seq* und *distinct par* in Tabelle 7.13 gehen wir von einem geordneten Stream aus. Dadurch, dass die Reihenfolge auch beim parallelen Filtern auf individuelle Streamelemente erhalten bleiben soll, braucht die parallele *distinct* Operation in etwa doppelt so lange wie die sequenzielle. Wenn wir unseren Stream hingegen über die Operation

unordered als ungeordnet deklarieren, kann die parallele Durchführung schneller geschehen. Das Problem bei geordneten Streams ist, dass es *distinct* darum geht, in welcher Reihenfolge die Elemente entdeckt wurden. Somit bleibt das erste Element im Stream und alle weiteren Duplikate werden herausgefiltert. Dadurch müssen bei parallelen Streams relativ viele Informationen zwischengespeichert werden, um erkennen zu können welches Element taskübergreifend tatsächlich als erstes vorgekommen ist.

Ganz davon abgesehen sieht man hier einen wesentlichen Vorteil der Verwendung von Streams. Bei der Operation *distinct* brauchen wir uns keine Gedanken über die Implementierung machen. Wir wenden *distinct* auf den Stream an und doppelte Elemente werden herausgefiltert. Versucht man manuell über eine for-Schleife die doppelten Einträge zu entfernen, so kann man auch auf ineffiziente Lösungen stoßen. Dazu schauen wir uns zunächst folgende Umsetzung mittels Schleife und *List.contains* an.

```
public List<String> distinctFor() {
    List<String> result = new ArrayList<>();
    for(String string : arrayList) {
        if(!result.contains(string)) {
            result.add(string);
        }
    }
    return result;
}
```

Hier sehen wir die Implementierung über eine *for-each* Schleife. Dabei überprüfen wir ob das Element schon in unserer resultierenden Liste vorhanden ist. Wenn nicht fügen wir es der Liste hinzu. Duplikate werden so herausgefiltert. Das Problem dabei ist, dass wir die Liste n mal durchlaufen, wobei n die Anzahl der Elemente ist. Insgesamt ergibt dies eine Laufzeit von $O(n^2)$. Das zeigt sich auch in den Benchmark-Ergebnissen der ineffizienten distinct Methode. Eine effizientere Möglichkeit wäre ein Set zu erzeugen. Dort gibt es keine Duplikate.

```
public List<String> distinctForOptimized() {
    HashSet<String> hashSet = new HashSet<>();
    for(String string : arrayList) {
        hashSet.add(string);
    }
    return new ArrayList<>(hashSet);
]
```

Die erste Variante in Tabelle 7.14 mit der bereits angesprochenen schlechten Laufzeit benötigt bei einhunderttausend Elementen knappe fünfundzwanzig beziehungsweise dreizehn Sekunden. Die optimierte Variante läuft in etwa so schnell wie die sequenzielle

	HotSpot		IBM J9	
Methode	Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
distinct for 100k	25083	2855	13520	5200
distinct optimized 100k	3,472	0,156	2,831	0,237
distinct optimized 1M	38,728	1,843	29,613	0,264

Tabelle 7.14: Benchmark: distinct Schleifen, Stream-Quelle: Liste mit einhunderttausend beziehungsweise einer Million Strings

Variante von oben. Bei *Stream.distinct()* wird intern mit ConcurrentHashMaps gearbeitet um doppelte Elemente herauszufiltern[Orab]. In Hinsicht auf geordnete parallele Streams verhält es sich bei *forEach* und *forEachOrdered* ähnlich wie bei *distinct.*7.15. Sobald wir bei parallelen Streams die Reihenfolge beachten müssen, können wir keine allzu hohen Performance-Verbesserungen erwarten.

	HotSpot		IBM J9	
Methode	Zeit(ms)	Error+-(ms)	Zeit(ms)	Error+-(ms)
forEachSeq	4,276	0,532	3,660	0,415
forEachOrderedSeq	4,173	0,174	3,806	0,196
forEachPar	3,119	0,049	2,925	0,236
forEachOrderedPar	4,886	0,239	8,015	0,299

Tabelle 7.15: Benchmark: forEachOrdered, Stream-Quelle: Liste mit 1 Million Strings

7.3 Java und Scala

In diesem Unterkapitel wollen wir die Performance der Streams im Vergleich zu Scala Collections näher betrachten. Wie wir bereits im Unterkapitel über Scala gelesen haben, haben Streams in Scala eine andere Bedeutung als in Java. Außerdem gibt es keine parallelen Streams in Scala. Daher werden wir stattdessen Java Streams mit Scala Collections vergleichen. Scala Collections beinhalten nämlich in etwa dieselben Methoden, die wir auch auf Java Streams anwenden können. So können wir ebenfalls Elemente Filtern, Mappen, Flatmappen, Gruppieren und Reduzieren. Das Pendant zu parallelen Streams in Java sind parallele Collections in Scala. Des Weiteren gibt es so genannte *Views* auf Collections in Scala. Dadurch werden bei den einzelnen Operationen nicht jedes Mal neue Collections erzeugt. Des Weiteren ist die Auswertung bei Views verzögert und nicht strikt, wie es bei sequenziellen und parallelen Collections der Fall ist. Views sind jedoch nur auf sequenzielle Collections anwendbar.

Da wir bereits in den vorherigen Unterkapiteln die Performance Unterschiede zwischen Hotspot und J9 veranschaulicht haben, werden in diesem Unterkapitel alle Benchmarks mit Hilfe der HotSpot JVM durchgeführt, da diese die Referenz-Implementierung darstellt. Für den Vergleich mit Scala verwenden wir *Scala 12.6*. In den folgenden Benchmarks

verwenden wir als Quelle des Streams vorzugsweise *ArrayList* in Java und *ArrayBuffer* beziehungsweise *Vektoren* in Scala, da dort die Performance wenig beeinflusst wird, wie wir für Java bereits im vorherigen Kapitel gesehen haben. Arrays wären hier nochmal etwas schneller, wie wir in Tabelle 7.8 gesehen haben, jedoch werden ArrayListen häufiger verwendet als Arrays.

Mapping Hier Mappen wir Zahlen von eins bis einer Million auf das Doppelte der jeweiligen Zahl. Anschließend geben wir eine Liste mittels *toList* als Ergebnis zurück. In Scala geben wir bei strikter Auswertung das Ergebnis der Berechnung zurück. Bei Views müssen wir die Berechnung wie bei Java Streams mit einer terminierenden Operation anstoßen beziehungsweise mit *force* in eine Collection umwandeln.

```
@Benchmark
def seq: Any = {
    return arrayBuffer.map(x => x*2)
}

@Benchmark
def par(): Any = {
    return arrayBuffer.par.map(x => x*2)
}

@Benchmark
def view(): Any = {
    return arrayBuffer.view.map(x => x*2).force
}
```

Methode	Zeit(ms)	Error+-(ms)
java sequenziell	11,655	0,522
java parallel	8,334	0,216
scala arraybuffer sequenziell	6,585	0,349
scala arraybuffer parallel	3,676	0,282
scala arraybuffer view	9,505	0,061

Tabelle 7.16: Benchmark: map, Stream-Quelle: ArrayListe/ArrayBuffer mit 1 Million Integern

Bei diesem Benchmark in Tabelle 7.16 ist Scala schneller als Java. Das liegt daran, dass wir bei Java explizit *toList()* angeben, um die Ergebnisse als Liste zurückzugeben. Bei Scala entfällt dies, da die Operationen auf Collections immer eine Collection als Resultat liefern und strikt ausgewertet werden. Schauen wir uns an was passiert, wenn wir in Scala jeweils ein Array oder eine Liste zurückgeben.

Methode	Zeit(ms)	Error+-(ms)
scala sequenziell toArray	8,700	0,510
scala parallel toArray	5,715	0,269
scala view toArray statt force	5,536	0,035
scala sequenziell toList	11,169	0,397
scala parallel toList	11,455	0,472
scala view toList statt force	17,041	1,710
scala sequenziell toVector	9,620	0,084
scala parallel toVector	8,047	1,815
scala view toVector statt force	7,297	0,067

Tabelle 7.17: Benchmark: Scala map toArray, N=1000000

Da die Listen in Scala als einfach verlinkte Listen (Tabelle 7.17) umgesetzt und unveränderlich sind, dauert es bei jener Variante mit *toList* länger. Für die nachfolgenden Tests werden wir jeweils einen *ArrayBuffer* als Input verwenden und als Resultat eine Collection vom Typ *Vector*. Die Verwendung von *Vector* ist ebenso wie die Verwendung von ArrayList weit verbreitet und Vector ist ebenfalls über einen Index zugreifbar. Somit fallen die Quelle und Resultat bei den Benchmarks nicht so sehr ins Gewicht.

Spezialisierte Streams Vergleichen wir nun Die Performance von Scala mit einem spezialisierten Stream in Java für den primitiven Typ *int*. Dazu nehmen wir das vorherige Beispiel und verwenden in Java anstelle von *map* die Methode *mapToInt* und erzeugen dadurch einen *IntStream* um automatisches Boxing und Unboxing bei der Addition zu vermeiden. Dabei sehen wir in Tabelle 7.18, dass Scala auf Grund von Boxing und Unboxing nicht mit der Performance von Java mithalten kann. Da wir in Java über das Mapping auf einen spezialisierten IntStream umwandeln, ersparen wir uns hier das automatische Boxing und Unboxing der Integer-Werte. In Scala haben wir dazu kein Pendant zum Stream mittels primitiven Typen.

Methode	Zeit(ms)	Error+-(ms)
Java sequenzieller Stream	4,959	0,082
Java paralleler Stream	0,980	0,036
Scala sequenzielle Collection	8,302	0,076
Scala parallele Collection	3,300	0,036

Tabelle 7.18: Benchmark: Java/Scala Boxing Unboxing, Mappen einer Million Integer und anschließend Summe ausgeben

Mehrere Zwischenoperationen Als nächstes wollen wir mehrere Operationen verketten. Da bei Scala eine strikte Auswertung erfolgt, wollen wir hier feststellen, inwiefern sich diese auf die Performance auswirkt, sobald wir mehrere Operationen durchführen.

In Scala werden in den einzelnen Zwischenschritten nämlich jedes mal neue Collections erzeugt. Außer es werden Views auf sequenzielle Collections angewandt. Wir führen hier mehrere Mapping Operationen durch, indem wir bei jeder Operation genau das gleiche Element zurückgeben. In Tabelle 7.19 sehen wir, dass bei einer Operation der Benchmark in Scala etwas langsamer als Java abläuft. Je mehr Zwischenoperationen durchgeführt werden, umso eher sehen wir, dass die strikte Auswertung der Collections in Scala zu einem Leistungsverlust führt. Dieser kann durch Anwendung von verzögert ausgewerteten Views eingedämmt werden und dann läuft der Benchmark in etwa so schnell wie in Java. In Hinsicht auf die parallele Verarbeitung der Mapping Operationen liegt Scala vor Java, da der parallele Stream in Java bei dieser Art von Benchmark kaum eine Verbesserung im Vergleich zum sequenziellen Stream bringt.

Anzahl Mapping	Methode	Zeit(ms)	Error+-(ms)
Eine Mapping Operation	Java sequenziell	1,792	0,037
	Java parallel	1,814	0,077
	Scala sequenziell	2,266	0,029
	Scala view	1,552	0,037
	Scala parallel	2,919	0,049
Fünf Mapping Operationen	Java sequenziell	4,546	0,344
	Java parallel	4,231	0,064
	Scala sequenziell	9,872	0,259
	Scala view	4,823	0,296
	Scala parallel	3,235	0,681
Zehn Mapping Operationen	Java sequenziell	8,091	0,140
	Java parallel	8,370	0,343
	Scala sequenziell	19,346	0,601
	Scala view	8,764	0,636
	Scala parallel	5,423	0,161

Tabelle 7.19: Benchmark: Java/Scala mehrere Mappings

Partitionierung Nun wollen wir uns unterschiedliche terminierende Operationen in Java und Scala anschauen. Als erstes Vergleichen wir hier collect mit *partitionBy*. Dabei teilen wir einhunderttausend beziehungsweise eine Million Studenten in männlich und weiblich auf. Die Testdaten sind so angelegt, dass jeweils fünfzig Prozent in die eine Kategorie fallen und fünfzig in die andere.

```
@Benchmark
public Map<Boolean, List<Student>> javaPartitionBySeq() {
  return studenten.stream()
    .collect(partitioningBy(
      student -> student.getGeschlecht().equals("m")));
}
```

```
@Benchmark
def scalaPartitionBy(): Any = {
  return studenten.
    partition(student => student.getGeschlecht.equals("m"));
}
```

Anzahl	Methode	Zeit(ms)	Error+-(ms)
Einhunderttausend	Scala sequenziell	1,594	0,094
	Scala parallel	0,662	0,022
	Java sequenziell	1,350	0,059
	Java parallel	0,919	0,044
Eine Million	Scala sequenziell	12,051	0,938
	Scala parallel	6,762	0,226
	Java sequenziell	14,221	2,426
	Java parallel	10,614	0,300
Zehn Millionen	Scala sequenziell	127,228	4,828
	Scala parallel	89,825	5,953
	Java sequenziell	146,408	6,050
	Java parallel	139,126	4,263

Tabelle 7.20: Benchmark: Java/Scala partitionBy

Die Ergebnisse des Benchmarks sehen wir in Tabelle 7.20. In Scala erhalten wir hier einen Tupel, bestehend aus beiden Partitionen als ArrayBuffer. Dabei ist in Scala vor allem die parallele Verarbeitung von *partitionBy* effizienter als in Java. Auch die sequenzielle Umsetzung ist hier in Scala schneller.

Gruppierung Hier werden Studenten nach Herkunftsland gruppiert. Je zehn Prozent werden auf zehn unterschiedliche Länder aufgeteilt. In Tabelle 7.21 sehen wir, dass Scala bei der sequenziellen Variante mit Java mithalten kann, jedoch bei zehn Millionen Elementen etwas langsamer abläuft. Hier haben wir bei den Tests allerdings mit vier beziehungsweise neun Millisekunden recht hohe Abweichungen. Parallel läuft die Gruppierung in Java schneller ab als in Scala. Hier kommt es immer auf die interne Implementierung der jeweiligen Operation an.

Reduce Minimum Im nächsten Beispiel suchen wir den besten Notendurchschnitt aller Studenten. Dazu verwenden wir die Methode *min* beziehungsweise *minBy*. Die parallele Verarbeitung in Scala bringt hier in Tabelle 7.22 außer bei einhunderttausend Elementen keinen Performance-Vorteil. Java ist hier eindeutig schneller als Scala.

Anzahl	Methode	Zeit(ms)	Error+-(ms)
Einhunderttausend	Scala sequenziell	1,984	0,062
	Scala parallel	1,838	0,015
	Java sequenziell	2,058	0,264
	Java parallel	1,077	0,123
Eine Million	Scala sequenziell	19,524	1,489
	Scala parallel	17,578	0,550
	Java sequenziell	19,459	1,199
	Java parallel	11,740	0,184
Zehn Millionen	Scala sequenziell	222,018	9,063
	Scala parallel	230,666	11,274
	Java sequenziell	214,400	4,826
	Java parallel	144,293	3,068

Tabelle 7.21: Benchmark: Java/Scala groupBy auf zehn Länder

```
public Optional<Student> findeBestenNotendurchschnittJava() {
  return studenten.stream().min(
    Comparator.comparingDouble(Student::getNotendurchschnitt));
}

def findeBestenNotendurchschnittScala(): Any = {
  return studenten.minBy(student => student.getNotendurchschnitt)
}
```

Anzahl	Methode	Zeit(ms)	Error+-(ms)
Einhunderttausend	Scala sequenziell	1,335	0,058
	Scala parallel	0,872	0,004
	Java sequenziell	1,294	0,086
	Java parallel	0,407	0,029
Eine Million	Scala sequenziell	10,645	0,056
	Scala parallel	10,435	0,170
	Java sequenziell	7,253	0,109
	Java parallel	5,100	0,270
Zehn Millionen	Scala sequenziell	104,484	1,791
	Scala parallel	112,747	5,393
	Java sequenziell	65,600	0,511
	Java parallel	46,788	0,511

Tabelle 7.22: Benchmark: Java/Scala min Notendurchschnitt

Reduce mit Summe Als nächstes *reduce* Beispiel wollen wir die bestandenen Prüfungen aller Studenten zählen. Dazu verwenden wir die Methode *sum*. Im Vergleich zum vorherigen Beispiel greift bei Tabelle 7.23 die Performance-Verbesserung bei Scala wieder. Java ist hier bei jeder Variante schneller als Scala. So ist die sequenzielle Berechnung der Summe in Java ab einer Million Elemente dreimal so schnell als in Scala. Auch die parallele Verarbeitung ist fast doppelt so schnell wie in Scala.

Anzahl	Methode	Zeit(ms)	Error+-(ms)
Einhunderttausend	Scala sequenziell	2,083	0,193
	Scala parallel	0,714	0,027
	Java sequenziell	1,410	0,022
	Java parallel	0,324	0,016
Eine Million	Scala sequenziell	15,444	0,846
	Scala parallel	7,819	0,020
	Java sequenziell	6,057	3,577
	Java parallel	3,577	0,009
Zehn Millionen	Scala sequenziell	144,332	2,485
	Scala parallel	79,479	1,344
	Java sequenziell	58,690	1,960
	Java parallel	36,608	0,062

Tabelle 7.23: Benchmark: Java/Scala Summe bestandener Prüfungen

flatMap In Tabelle 7.24 wollen wir die Stream-Operation *flatmap* innerhalb von Java und Scala vergleichen. Dabei wollen wir alle Zeugnisse aller Studenten zurückbekommen. Jeder Student hat dabei einhundert Zeugnisse. Dabei sehen wir, dass die parallele Verarbeitung bei Verwendung von *flatmap* in Scala langsamer als die sequenzielle Verarbeitung der Daten ist. Der zusätzliche Berechnungsaufwand für die parallele Variante macht diese ineffizient. Sequenziell ist Scala hier schneller als Java.

Anzahl	Methode	Zeit(ms)	Error+-(ms)
Zehntausend Studenten	Scala sequenziell	6,005	0,483
	Scala parallel	9,539	0,320
	Java sequenziell	9,336	0,302
	Java parallel	5,829	0,149
Einhunderttausend Studenten	Scala sequenziell	72,297	0,699
	Scala parallel	100,783	1,343
	Java sequenziell	103,850	5,795
	Java parallel	89,369	1,366

Tabelle 7.24: Benchmark: Java/Scala flatMap, Pro Student einhundert Zeugnisse

7.4 Ergebnisse

Vergleich zwischen Schleifen und Streams Bei den Benchmarks, die direkt den Vergleich zwischen Schleifen und Streams dargestellt haben, sehen wir, dass meistens die alte Variante mit den Schleifen etwas schneller als jene mit sequenziellen Streams abläuft. Grund dafür ist der zusätzliche Aufwand, der über die Erzeugung des Streams und dessen Verarbeitung der Operationen zustande kommt. Dennoch sieht man meist nur sehr geringe Laufzeit-Unterschiede zwischen Schleifen und Streams. Hier muss man bedenken, dass die Realisierung der Streams auf Grund der flexiblen Implementierung mittels *invokedynamic* in Zukunft noch effizienter umgesetzt werden kann. Auch in Hinsicht auf JIT-Optimierungen ist das Konzept der Streams API relativ neu. Bei zukünftigen Versionen kann man hier noch Performance-Verbesserungen erwarten. Hier muss man auch noch erwähnen, dass Performance in Hinsicht auf die Streams API nicht immer das wichtigste ist, wie wir in Kapitel 4 gesehen haben. Als allgemeine Fallstricke bei sequenziellen und parallelen Streams ist Boxing und Unboxing zu beachten. Hier sollte man bei Berechnungen immer einen IntStream als Quelle verwenden, beziehungsweise über Operationen wie *mapToInt* auf die spezialisierte, für primitive Datentypen geeignete Stream-Variante wechseln, da man ansonsten mit Performance-Einbußen zu rechnen hat.

Parallele Streams Wie wir gesehen haben, lohnt sich die Verwendung von parallelen Streams im Vergleich zu sequenziellen Schleifen beziehungsweise sequenziellen Streams durchaus. Wichtig dabei ist, dass wir eine geeignete Quelle für den Stream verwenden, wie beispielsweise ein Array oder Array-Listen. Weniger gut eignen sich Dateien, verlinkte Listen und synchronisierte Methoden. Die synchronisierte Methode ist sequenziell um das Zehnfache schneller als bei parallelen Streams. Des Weiteren muss man auch auf die Collection beziehungsweise die terminierenden Operationen achten, über welche letztlich die Teilergebnisse der parallelen Streams zusammengefügt werden. Auch hier kann man einiges an Performance einbüßen. So ist ein paralleles collect zu einem Set oder einer Map in etwa dreimal so langsam wie das sequenzielle Äquivalent. Stattdessen sollte man entweder Listen verwenden oder eine *concurrent Map*, welche für gleichzeitige Zugriffe geeignet ist. Auch die Methode *joining* ist für die parallele Verarbeitung auf Grund der Unveränderlichkeit von Strings nicht geeignet.

Eine noch wichtigere Rolle spielt bei parallelen Streams jedoch der Aufwand, den die einzelnen Tasks für die Berechnung der Operationen benötigen. Dafür relevant ist neben der Anzahl der Elemente vor allem die Komplexität der durchgeführten Berechnungen. Auch die erwähnten Fallstricke im Zusammenhang mit parallelen Streams sind zu beachten. Dazu zählt der allgemeine Fehler keine spezialisierten Streams für primitive Datentypen zu verwenden. Außerdem muss man bei zustandsbehafteten Zwischenoperationen, wie *distinct* und *sorted* darauf achten, einen nicht geordneten Stream zu verwenden. Bei geordneten Streams mit zustandsbehafteten Zwischenoperationen sollte man daher sequenzielle Streams verwenden.

Vergleich zwischen Hotspot JVM und IBM J9 Bei den durchgeführten Benchmarks im Zusammenhang mit Schleifen war die Hotspot JVM etwas schneller als die IBM J9. Bei sequenziellen Streams war der Performance-Unterschied zwischen den beiden getesteten JVMs relativ gering. So war bei einigen Benchmarks die J9 schneller und bei anderen die Hotspot JVM. Beim *grouping* und *flatmap* war die Hotspot JVM um etwa dreißig Prozent schneller als die J9. Hingegen war beim Benchmark auf unterschiedliche Stream-Quellen die J9 schneller. Bei der parallelen Verarbeitung konnte die J9 größtenteils überzeugen. Bis auf die Benchmarks mit *flatMap* und *collect* war die J9 vor allem im Umgang mit vielen Elementen schneller als die Hotspot JVM. Trotzdem sieht man hier, dass die Performance-Unterschiede zwischen Hotspot JVM und IBM J9 sehr situationsabhängig sind. Wenn man Performance-kritische Applikationen einsetzt und die Verwendung einer JVM außerhalb der Referenzimplementierung der Hotspot JVM nicht ausschließt, so bleibt einem die Durchführung von Benchmarks der jeweiligen Applikation nicht erspart.

Vergleich zwischen Java und Scala Der Vergleich zweier Programmiersprachen in Hinsicht auf deren Laufzeit-Performance ist nicht trivial. Einerseits vergleichen wir hier mehr oder weniger unterschiedliche Konzepte. In Java verwenden wir Streams und in Scala werden Collections verwendet, wobei Streams verzögert und Collections strikt ausgewertet werden. Auch auf die unterschiedliche Umsetzung der Datentypen muss bei den Benchmarks geachtet werden, wie wir beispielsweise bei Listen gesehen haben. Bei Java und Scala haben wir immerhin noch den Vorteil, dass beide auf der JVM laufen und eine Interoperabilität zwischen den beiden Sprachen gegeben ist. Außerdem wird für die parallele Umsetzung dasselbe Fork-Join-Framework verwendet.

Vor allem wenn es um automatisches Boxing und Unboxing von primitiven Typen auf Referenztypen geht, kann Scala in Hinsicht auf die Laufzeit-Performance von Java nicht mithalten. Auch bei der Verwendung von mehreren Zwischenoperationen hat man bei Scala auf Grund der strikten Auswertung bei sequenziellen Collections einen Nachteil. Dieser lässt sich jedoch mit Hilfe von Views umgehen.

Bei den sonstigen Benchmarks, die zwischen Java und Scala durchgeführt wurden, war es relativ ausgeglichen. So war beispielsweise beim Mappen und anschließendem *collect* Scala schneller als Java. Auch bei der Partitionierung war Scala sowohl sequenziell als auch parallel vorne. Beim Gruppieren waren beide in sequenzieller Hinsicht gleich auf, jedoch konnte Scala hier parallel keine schnellere Gruppierung der Studenten durchführen. Daher war Java parallel fast doppelt so schnell. Beim Suchen des Minimums und bei der Berechnung der Summe war Java sequenziell und parallel schneller als Scala. Bei *flatmap* sehen wir interessanterweise, dass Scala sequenziell schneller ist. Jedoch kann die Operation nicht effizient parallel ausgeführt werden. Hier ist bei Scala die parallele Abarbeitung langsamer als die sequenzielle. Hier muss man jedoch dazusagen, dass die sequenzielle Verarbeitung in Scala bei zehntausend Studenten nur etwas langsamer abläuft und bei einhunderttausend Studenten mit je einhundert Zeugnissen sogar zwanzig Prozent schneller verarbeitet werden kann als die parallele Variante in Java.

KAPITEL 8

Vergleich der Konzepte: Java und Haskell

In diesem Kapitel soll überprüft werden, welche Konzepte der funktionalen Programmierung durch das Hinzufügen von Lambda-Ausdrücken und Streams in Java 8 ermöglicht, beziehungsweise zu Vorgängerversionen vereinfacht wurden und welche Konzepte nach wie vor nur mit zusätzlichem Aufwand umsetzbar sind, oder auf Grund der Gegebenheiten in Java eher unpraktisch anwendbar sind. Dazu werden wichtige funktionale Konzepte der Programmiersprache Haskell herangezogen, um dadurch einen Vergleich der Umsetzbarkeit in Java durchführen zu können. Der Fokus liegt hier auf der Umsetzung der Konzepte in Java. So werden wir unter anderem unveränderliche Datentypen, Pattern-Matching und algebraischen Datentypen in Java 8 vorstellen.

8.1 Algebraische Datentypen und Pattern-Matching

In der funktionalen Programmierung spielen algebraische Datentypen und Pattern-Matching eine wichtige Rolle. Mit Hilfe von Pattern Matching können bei Funktionen, je nachdem welche Struktur die Eingabeparameter haben, unterschiedliche Funktionskörper ausgeführt werden. In der objektorientierten Programmierung wird stattdessen dynamisches Binden eingesetzt um Pattern-Matching umzusetzen. Dynamisches Binden wird in diesem Unterkapitel noch genauer vorgestellt. Pattern-Matching verhält sich ähnlich wie die *switch-case* Anweisung in Java. Das nachfolgende Beispiel präsentiert das sehr simpel gehaltene Pattern-Matching in Java.

```
switch(x) {
    case "a":
        System.out.println("Zweig a wird ausgeführt");
        break;
```

```
        case "b":
            System.out.println("Zweig b wird ausgeführt");
            break;
        default:
            System.out.println("Der Standardzweig wird ausgeführt");
            break;
}
```

In dieser *switch-case* Anweisung wird der Zweig, der ausgeführt werden soll, über die Variable x, welche in diesem Beispiel den Typ *String* hat, bestimmt. Dabei werden drei unterschiedliche Fälle angelegt. Einer für den Fall, dass die Variable *x* gleich dem String *a* entspricht, einer für *b*. Sollte keines dieser Muster zutreffen, so wird der standardmäßig zu verwendende Zweig, der über das Schlüsselwort *default* festgelegt ist, ausgeführt. Diese Art des Pattern-Matchings könnte in Java auch mit *if*, *elseif* und *else* durchgeführt werden. Dabei hätte man mehr Möglichkeiten im Vergleich zu *switch-case*, da bei *switch-case* nur über die Variable im *switch* die anzuwendenden Fälle zu unterscheiden sind. Innerhalb der Anweisung *switch* dürfen außerdem nur die primitiven Datentypen *byte*, *short*, *char* und *int* verwendet werden und zugehörige boxed Typen. Zusätzlich dazu dürfen noch *Enum-Typen* und seit Java 7 *Strings* verwendet werden. Eine weitere Möglichkeit für Pattern-Matching in Java ist über die Verwendung von Exceptions im Zusammenhang mit try-catch möglich. Hier entspricht jede catch-Klausel einem case[Pun07].[Oraf]

Als nächstes wollen wir Pattern-Matching in Haskell näher betrachten. Dazu wird folgendes Beispiel verwendet. Es handelt sich dabei um *map*, welche mit der Stream-Operation *map* von Java 8 vergleichbar ist.

```
map _ []     = []
map f (x:xs) = f x : map f xs
```

Hier sehen wir, dass bei der Funktion *map* die Eingabeparameter über die Auswahl des Funktionskörpers entscheiden. Die Funktion *map* nimmt eine Funktion und eine Liste entgegen. Beim Pattern-Matching in Haskell wird zwischen vier Mustern unterschieden. Der Unterstrich _ steht für eine so genannte *Wildcard*. Dabei kann eine beliebige Eingabe verwendet werden. Der Eingabeparameter wird bei der Ausführung der Funktion, also im Funktionskörper, nicht mehr benötigt. Die eckigen Klammern [] stehen für eine leere Liste. In der zweiten Zeile wird das zweite Muster für *map* festgelegt. Dabei steht *f* ebenfalls für einen beliebigen Eingabeparameter, welcher die Eingabe an den Parameter f bindet. Der Wert *x:xs* repräsentiert eine nicht-leere Liste. Hierbei steht das *x* für das erste Element der Liste, welcher auch als Kopf der Liste bezeichnet wird. Das *xs* bezieht sich den Rest der Liste ohne den Listenkopf *x* und wird als *Listenrest* bezeichnet. In Haskell bezieht sich der Operator : auf die Verkettung einer Liste. Über Pattern-Matching werden nun folgende Funktionen für *map* festgelegt. Wenn wir der Funktion eine leere Liste übergeben, so ist es irrelevant, welche Funktion wir an *map* übergeben. Es wird

immer die leere Liste zurückgeliefert, da keine Elemente für ein Anwenden der Funktion verfügbar sind. Wenn wir an *map* eine Funktion *f* und eine nicht-leere Liste übergeben, so wird *f* auf das erste Element der Liste angewandt. Dieser Funktionsaufruf ist in Haskell über *f x* dargestellt. Die Funktion wird also nicht wie in Java über runde Klammern aufgerufen. Das Resultat der Funktion bildet gemeinsam mit dem rekursiven Aufruf von *map* mit der selben Funktion *f* und dem Listenrest das Ergebnis von *map*. Die Funktion wird also so lange auf die Elemente der Liste aufgerufen bis der Listenrest eine leere Liste repräsentiert und die rekursive Funktion *map* dadurch terminieren kann.[HPF99]

Pattern-Matching in Haskell kann aber nicht nur auf Werte oder Listen angewandt werden. Es funktioniert auch im Zusammenhang mit algebraischen Datentypen. Leider wurden diesbezüglich in Java 8 keine neuen Pattern-Matching Mechanismen hinzugefügt. Sofern man keine externen Bibliotheken verwenden will, muss man sich hier nach wie vor mit *if*-Anweisungen oder für einfache Muster mit *switch-case* begnügen. Innerhalb dieses Unterkapitels sehen wir jedoch, dass wir zumindest eine verbesserte Variante von Pattern-Matching in Java mit Hilfe von Lambda-Ausdrücken umsetzen können.

Nun wollen wir uns mit algebraischen Datentypen beschäftigen. Zuerst sollen diese und deren Vorteile über Haskell erklärt werden. In diesem Zusammenhang wird auch nochmals auf das Pattern-Matching näher eingegangen. Danach schauen wir uns die Unterschiede zwischen algebraischen Datentypen und Interfaces und Klassen an. Danach werden wir sehen, welche neuen Möglichkeiten es für Pattern Matching und algebraische Datentypen in Java 8 gibt. Algebraische Datentypen setzen sich aus ein oder mehreren Datentypen zusammen. So können sie einerseits eine Aufzählung von möglichen Typen sein, wie beispielsweise ein *Boolean*, der aus den Typen *True* und *False* bestehen kann. Des Weiteren können sie verwendet werden um Daten in einem Datentyp zu kapseln (Tupel). Außerdem lassen sich damit rekursive Strukturen wie Listen oder Bäume definieren. Für dieses Unterkapitel verwenden wir den folgenden Baum um anhand von Beispielen algebraische Datentypen in Haskell und Java vorzustellen. Dieser ist in Haskell-Syntax angegeben.[JPK10]

```
data Tree a = Leaf a
            | Branch a (Tree a) (Tree a)
            | Empty
```

Hier erzeugen wir den algebraischen Datentypen *Tree*. Dieser setzt sich aus den folgenden Typen zusammen: *Leaf*, *Branch* oder *Empty*. Der Typ *Branch* ist dabei rekursiv definiert. Er beinhaltet zwei weitere Teil-Bäume. Auf dieser neuen Struktur wollen wir nun Operationen definieren. Für die Beispiele in diesem Unterkapitel verwenden wir den folgenden Baum.

Dieser würde in Haskell folgendermaßen aussehen:

```
Branch 1 (Branch 2 (Leaf 3) (Leaf 4)) (Branch 3 (Leaf 6) (Empty)
```

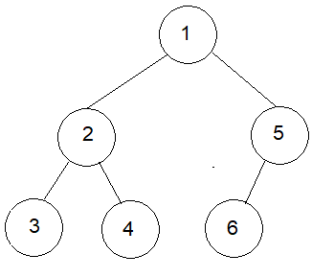


Abbildung 8.1: Beispiel: Baum

Mit Hilfe von Pattern Matching können wir in Haskell relativ einfach Operationen auf diesen algebraischen Datentypen definieren. Dabei wollen wir beispielsweise die Summe des Baumes berechnen.

```
calcSum :: Num a => Tree a -> a
calcSum (Leaf x) = x
calcSum (Branch x l r) = x + calcSum l + calcSum r
calcSum (Empty) = 0
```

Für ein Blatt verwenden wir den Wert des Blattes, für ein leeres Blatt *Empty* wird der Wert null zurückgegeben. Die Berechnung für den Wert eines Zweiges ist rekursiv. Der Wert des Zweiges selbst und alles was links und rechts dranhängt werden addiert. Beim Pattern-Matching in Haskell wird dabei auf Vollständigkeit geachtet. Fehlende Muster werden vom Compiler erkannt, sofern man *-fwarn-incomplete-patterns* als Parameter für die Kompilierung angibt. Dadurch kommt, wenn man zum Beispiel *Empty* als Muster weglässt folgende Fehlermeldung.

```
Pattern match(es) are non-exhaustive
In an equation for `calcSum': Patterns not matched: Empty
```

Wir verwenden hier einen in sich geschlossenen Datentypen. Das heißt wir definieren einmal die Struktur und diese bleibt unverändert. Mit Hilfe von Pattern-Matching ist es einfach neue Operationen hinzuzufügen. Die Struktur des Datentypen zu ändern wäre jedoch ein größerer Aufwand, da wir alle dafür definierte Operationen an diese neue Struktur anpassen müssten.

8.1.1 Umsetzung mit objektorientierter Programmierung

Bevor wir uns ansehen ob und inwiefern Java algebraische Datentypen unterstützt, sehen wir uns die Umsetzung des Beispiels über objektorientierte Programmierung in Java an. Dazu verwenden wir Vererbung.

```
public interface Tree {
    Integer add();
}

public class Branch implements Tree {
    public int value;
    public Tree left;
    public Tree right;

    public Branch(int value, Tree left, Tree right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
    @Override
    public Integer add() { return value + left.add() + right.add(); }
}

public class Leaf implements Tree {
    public int value;

    public Leaf(int value) { this.value = value; }
    @Override
    public Integer add() { return value; }
}

public class Empty implements Tree {
    @Override
    public Integer add() { return 0; }
}
```

Um den algebraischen Datentypen *Tree* in Java zu simulieren, verwenden wir ein Interface. Die dadurch zusammengesetzten Typen werden über Klassen, welche das Interface implementieren umgesetzt. Das Interface definiert die Methode *add* die einen Integer zurückliefert. Die einzelnen Klassen müssen diese abstrakte Methode überschreiben. Dadurch können wir nun die Summe des gesamten Baumes berechnen. Folgender Code erzeugt unseren Beispielbaum in Java und berechnet dessen Summe.

```
Tree tree = new Branch(1,
                new Branch(2, new Leaf(4), new Leaf(5)),
                new Branch(3, new Leaf(6), new Empty()));

tree.add liefert 21
```

Bei objektorientierter Programmierung ist es hier über die eingesetzte Vererbung relativ einfach neue Untertypen für unseren dargestellten algebraischen Datentypen hinzuzufügen. Beim Hinzufügen von neuen Operationen müssen wir jedoch alle Typen, welche für den algebraischen Datentypen verwendet wurden, ändern. Daher ist das Hinzufügen neuer Operationen aufwendig. Eventuell haben wir nicht den benötigten Zugriff auf diese Klassen um überhaupt neue Operationen hinzuzufügen. Dies könnte zum Beispiel bei der Verwendung von Bibliotheken der Fall sein. Wir wollen nun versuchen die Operationen unabhängig von den Klassen zu definieren. Dazu schauen wir uns zunächst an wie Java algebraische Datentypen unterstützt.[PJ98]

8.1.2 Algebraische Datentypen in Java

Algebraische Datentypen werden in Java über *Enums* mehr oder weniger unterstützt. Enums sind dabei die Abkürzung für *Enumeration* beziehungsweise Enumeration Type. Damit sind also Aufzählungen von Werten gemeint. Diese werden auch als Summen-Typen bezeichnet. Enums wurden in Java 5 hinzugefügt. Davor konnte man in Java Konstanten verwenden um Aufzählungen zu simulieren. Nachfolgend sehen wir unseren algebraischen Datentypen als Integer-Konstanten umgesetzt.[GL96]

```
public class Tree {
    public static final int EMPTY = 0;
    public static final int LEAF = 1;
    public static final int BRANCH = 2;
}
```

Leider bietet uns diese Variante keine Typ-Sicherheit. Ganz davon abgesehen, dass wir den Komponenten des Baumes so auch keine Werte zuweisen können. Folgendermaßen könnten wir unseren Baum als Enumerations-Typen ab Java 5 definieren.

```
public enum Tree {
    BRANCH, LEAF, EMPTY;
}
```

Hiermit hätten wir eine typ-sichere Möglichkeit unseren Baum umzusetzen. Wir können bei diesem Enum über *switch case* Pattern Matching anwenden. Außerdem sind Enum in Java nicht erweiterbar. Das heißt wir haben hier einen abgeschlossenen Typ erzeugt. Enums sind in Java ein Spezialfall von Klassen. Da sie als Objekte umgesetzt sind, können diese auch Methoden enthalten[GJS$^+$15]. In Klammer können wir den Enums auch konstante Werte übergeben. Leider können wir keine Enums für die Umsetzung unseres Beispiels verwenden. „An enum type has no instances other than those defined by its enum constants. It is a compile-time error to attempt to explicitly instantiate an enum type (§15.9.1)“[GJS$^+$15] Daher können wir keine unterschiedlichen Instanzen für unseren Baum erzeugen.

Wir können nun Versuchen den Enum-Typen für unseren Baum in einer markierten Klasse (*tagged class*) zu verwenden. Dabei haben wir ein Feld, über das wir quasi den Typ der Klasse deklarieren. Dazu erzeugen wir folgende Klasse:

```
public class TreeClass {
    private Tree type;
    private Integer value;
    private TreeClass left;
    private TreeClass right;
}
```

Zusätzlich würden wir den Konstruktor noch als privat festlegen. Außerdem benötigten wir unterschiedliche Methoden für die Erzeugung der TreeClass-Objekte.

```
public static TreeClass instanceOfLeaf(int value) {
    TreeClass treeClass = new TreeClass();
    treeClass.type = Tree.LEAF;
    treeClass.value = value;
    return treeClass;
}
```

Dann können wir in einer switch Anweisung die Summe auf Grund des markierten Typen der Klasse *type* berechnen.

```
public static int sum(TreeClass treeClass) {
  switch (treeClass.getType()) {
    case BRANCH:
      return treeClass.getValue() + sum(treeClass.getLeft())
        + sum(treeClass.getRight());
    case LEAF:
      return treeClass.getValue();
    case EMPTY:
      return 0;
    default:
      return 0;
  }
}
```

Schauen wir uns nun die Probleme dieses Ansatzes an. Wir benötigen für die Erstellung der Klasse viel unnötigen Code. Dazu zählt die Deklaration des *enums*, das *type* Feld in unserer Baumklasse. Zusätzlich dazu haben wir mehrere statische Methoden um die

unterschiedlichen Objekte zu erzeugen, da jedes Objekt andere Felder gesetzt bekommen muss. Die Anweisung *switch* achtet nicht auf Vollständigkeit. Daher kann es zu Laufzeitfehlern kommen, wenn wir einen *case* vergessen, beziehungsweise da wir *default* gesetzt haben, würde es hier zu fehlerhaften Berechnungen führen wenn wir einen Fall vergessen. Wir benötigen auch mehr Speicherplatz, da jeder simulierte Typ alle Felder beinhaltet. Außerdem verwenden wir hier keine Klassen-Hierarchie, sondern imitieren diese nur über die Verwendung von *enum*.[Blo08]

8.1.3 Alternative Möglichkeiten

Eine weitere Möglichkeit wäre es, dem Interface Tree eine Methode default-Methode für die Berechnung der Summe des Baumes hinzuzufügen. Um die Unabhängigkeit zu erhöhen, könnte man diese Methode auch als statische Methode in eine eigene Klasse auslagern.

```
default Integer sum(Tree tree) {
    int sum = 0;

    if (tree instanceof Leaf) {
        Leaf leaf = (Leaf) tree;
        sum += ((Leaf) tree).value;
    } else if (this instanceof Branch) {
        Branch branch = (Branch) tree;
        sum += branch.value + sumWithCasts(branch.left) +
                sumWithCasts(branch.right);
    }

    return sum;
}
```

Dabei wird zur Laufzeit überprüft ob es sich bei dem Objekt um ein Blatt oder einen Zweig handelt. Nach der Überprüfung wird das Objekt von *Tree* auf den jeweiligen Typen *Leaf* oder *Branch* umgewandelt. Das Problem bei dieser Variante sind die Verwendung von *instanceof* und Typumwandlungen (Typecasts). Beide werden als so genannte Code-Smells angesehen und dessen Verwendung sollte weitgehend vermieden werden. Bei der expliziten Typumwandlung kann der Compiler statisch keine Probleme feststellen. Dadurch kann es bei Typ-Umwandlungen zu Fehlern kommen, welche jedoch erst zur Laufzeit entdeckt werden und zum Absturz des Programmes führen können[VEM02]. Außerdem haben wir hier auch keine Überprüfung ob auch wirklich alle Fälle des algebraischen Datentypen abgedeckt sind. Als nächstes versuchen wir über eine neue Klasse die Operation auszulagern. Dabei wollen wir ohne *instanceof* und Typ-Umwandlungen auskommen.[BRLG04]

```
public class Add {
  public int add(Tree tree) {
    throw new IllegalStateException("Sollte nicht augerufen werden");
  }

  public int add(Leaf leaf) {
    return leaf.value;
  }

  public int add(Branch branch) {
    return branch.value + add(branch.left) + add(branch.right);
  }

  public int add(Empty empty) {
    return 0;
  }
}
```

Wir erzeugen vier überladene Methoden *add*. Für jeden Typen, aus dem unser Baum zusammengesetzt ist, wird eine andere *add* Methode verwendet. Zur Laufzeit haben die einzelnen Teile des Baumes nie den dynamischen Typ Baum. Die Methode sollte daher nur für Blätter und für Zweige aufgerufen werden. Daher könnten wir die Methode *add(Tree tree)* eigentlich weglassen. Doch dann lässt sich unsere Klasse nicht mehr kompilieren, da wir in *add branch* über *branch.left* und *branch.right* die Methode *add tree* aufrufen. Daher haben wir die Methode hinzugefügt. Versucht man nun für unseren Beispielbaum die Summe über die Klasse *Add* zu berechnen, bekommen wir folgendes Ergebnis zurück.

```
Exception in thread "main" java.lang.IllegalStateException:
    Sollte nicht augerufen werden
```

Die Summe konnte nicht berechnet werden, da *add(Tree tree)* aufgerufen wurde, obwohl wir eigentlich davon ausgegangen sind, dass diese Methode auf Grund der dynamischen Typen nie aufgerufen werden kann. Dieser Lösungsansatz ist in Java nicht möglich. Der Grund dafür ist, dass Java keine Multi-Methoden, welche über mehrfach dynamisches Binden realisiert werden, unterstützt. Java unterstützt nur einfaches dynamisches Binden auf das Objekt, welches die Methode aufruft, nicht jedoch auf die Parameter.

```
Tree tree = new Leaf(1);
tree.doSomething();
```

Die Variable tree hat hier den statischen Typ *Tree*. Der dynamische Typ zur Laufzeit ist jedoch *Leaf*. Da Java dynamisches Binden für den Aufrufer unterstüzt würde die Methode

doSomething() hier wie erwartet in der Klasse *Leaf* ausgeführt werden. Das Gleiche gilt jedoch nicht für die Parameter einer Methode. Hier werden nur die statischen Typen betrachtet. In unserem Beispiel haben *branch.left* und *branch.right* den statischen Typ *Tree*. Daher wird *add(branch.left)* beziehungsweise *add(branch.right)* immer *add(Tree tree)* ausführen, obwohl der dynamische Typ zur Ausführung immer einer der folgenden ist: *Empty*, *Leaf* oder *Branch*. Daher kommt in Sprachen ohne Multi-Methoden das *Visitor Pattern* zum Einsatz.[Pun07][CLCM00]

8.1.4 Visitor Pattern

Das Visitor Pattern ist eines der klassischen Entwurfsmuster der objektorientierten Programmierung. Es wird dazu verwendet, um neue Operationen auf ein Objekt zu ermöglichen, ohne dabei die Klassen des Objekts selbst ändern zu müssen. Dies wird dadurch erreicht, dass die Operationen nicht in den jeweiligen Klassen, auf welche sie angewandt werden sollen, implementiert werden. Stattdessen wird das so genannte *Visitor Pattern* verwendet um die Operationen auszulagern. Das Visitor Pattern versucht das Problem der objektorientierten Programmierung im Zusammenhang mit algebraischen Datentypen zu lösen: Die Erweiterung um neue Operationen ist einfach. Das Hinzufügen neuer Untertypen ist aufwendiger, da hier alle Operationen des Besuchers ebenfalls neu implementiert werden müssen.[PJ98]

Dabei geht es um die Trennung von Verhalten (Operationen) und Zustand (strukturbildende Klassen). Klassen und Operationen darauf sollen unabhängig voneinander sein. Dadurch wird die Wiederverwendbarkeit der Klassen einerseits erhöht. Andererseits bleiben die Klassen beim Hinzufügen neuer Operationen unverändert. Ein Weiterer Punkt bezieht sich auf verwendete Frameworks und Bibiliotheken. Da diese meist von anderen Entwicklern sind, kann man die Struktur der Klassen nicht ändern. Mit Hilfe des Besucher-Entwurfmusters ist es dennoch möglich eigene Operationen hinzuzufügen. Für die Darstellung des Entwurfsmusters verwendet wir wieder folgendes Beispiel einer algebraischen Datenstruktur in Haskell-Syntax.[BRLG04]

Um die strukturellen Klassen und die Operationen voneinander unabhängig zu machen, können wir nun das Besucher-Entwurfsmuster einsetzen. Dazu erzeugen wir ein Interface mit dem Namen *Visitor*. Für jede Operation erzeugen wir eine Klasse welche *Visitor* implementiert.

```
public interface Visitor {
    void visit(Branch branch);
    void visit(Leaf leaf);
    void visit(Empty empty);
}

public class AddVisitor implements Visitor {
    private int sum = 0;
```

```
    @Override
    public void visit(Branch branch) {
        sum = sum + branch.value;
        branch.left.visit(this);
        branch.right.visit(this);
    }

    @Override
    public void visit(Leaf leaf) {
        sum = sum + leaf.value;
    }

    @Override
    public void visit(Empty empty) {
        // do nothing
    }
}
```

Im Interface definieren wir für jeden möglichen Typen von *Tree* eine überladene Methode *visit*. Für die Operation erzeugen wir eine Unterklasse des Visitors. Für *Empty* wird bei *visit* nichts gemacht. Für ein Blatt wird der Wert des Blattes zur Summe dazu addiert. Beim Zweig werden die visit Methoden der jeweiligen Teilbäume aufgerufen. Damit dieser Code funktioniert müssen wir noch in den einzelnen Klassen des Baumes die *visit* Methode implementieren.

```
// wird in Empty, Leaf und Branch implementiert
@Override
public void visit(Visitor visitor) {
    visitor.visit(this);
}

Main.java:
AddVisitor addVisitor = new AddVisitor();
tree.visit(addVisitor);
addVisitor.getSum();
```

Die überschriebene Methode sieht bei allen drei Klassen (Empty, Leaf, Branch) gleich aus. Dennoch ist es nicht möglich diese Methode als default-Methode im Interface *Tree* anzugeben und sie aus den Unterklassen zu entfernen. Wenn wir nun eine neue Operation hinzufügen wollen, so müssen wir lediglich eine neue Unterklasse wie *AddVisitor* implementieren. Warum funktioniert diese Variante mit Hilfe des Visitor Patterns, obwohl wir in Java keine Multimethoden und nur einfaches dynamisches Binden umgesetzt wird?

Durch Hinzufügen einer Indirektion können wir Multimethoden und damit mehrfach dynamisches Binden simulieren. In diesem Fall wird zwei mal dynamisch Gebunden. Zuerst wird über *visitor.visit(this)* die Methode *visit* über den dynamischen Typen – also jenen Typen zur Laufzeit – von *visitor* gebunden. Dadurch wird immer die richtige Operation ausgeführt, in unserem Fall *add* über den *AddVisitor*. Das zweite mal wird bei *visit(Branch branch)* dynamisch gebunden. In diesem Fall wird über den Typ von *branch.left* beziehungsweise *branch.right* zur Laufzeit die richtige *visit* Methode in Empty, Leaf oder Branch ausgeführt. Wenn man auf mehrere Parameter dynamisch Binden möchte, so muss man für jeden weiteren Parameter einen weiteren Schritt des einfachen dynamischen Bindens hinzufügen.[Pun07]

Das Visitor Pattern hat jedoch die folgenden Nachteile: Bestehender Code muss um die *visit* Methode erweitert werden, damit das Visitor Pattern funktionieren kann. Dies könnte bei Bibliotheken und Frameworks zu Problemen kommen. Des Weiteren kann die Anzahl der benötigten Methoden schnell sehr groß werden.[BRLG04] [Pun07]

8.1.5 Neue Möglichkeiten in Java 8

Schauen wir uns nun an, welche Möglichkeiten wir in Java 8 haben um Pattern-Matching beziehungsweise algebraische Datentypen umzusetzen. Es wurden keine direkten Verbesserungen für beide Themenbereiche in Java 8 zur Sprache hinzugefügt. Dennoch gibt es neue Mechanismen um Pattern-Matching und algebraische Datenstrukturen sinnvoller verwenden zu können.

Method-Handles Die im Kapitel der *JVM* angesprochenen Method-Handles wurden zwar bereits zu Java 7 hinzugefügt, verwendet werden sie jedoch erst seit Java 8 im Zusammenhang mit Lambda-Ausdrücken. Ebenso wie *invokedynamic*. So könnte man für die Auslagerung der Operationen auf algebraische Datenstrukturen Method-Handles verwenden. Dazu verändern wir den Code der im vorherigen Unterkapitel vorgestellten Methode *add(Branch branch)* der Klasse *Add*. Diese war auf Grund der fehlenden Multimethoden in Java nicht korrekt implementiert.

```
Add.java:
private MethodHandles.Lookup lookup = MethodHandles.lookup();

public int add(Tree tree) throws Throwable {
  MethodHandle methodHandle = lookup.findVirtual(
      AddMethodHandles.class, "add",
      MethodType.methodType(int.class, tree.getClass()));
  return (int)methodHandle.invoke(this,tree);
}

public int add(Branch branch) throws Throwable {
  MethodHandle left = lookup.findVirtual(
```

```
      Add.class, "add",
      MethodType.methodType(int.class, branch.left.getClass()));
  MethodHandle right = lookup.findVirtual(
      Add.class, "add",
      MethodType.methodType(int.class, branch.right.getClass()));

  return branch.value +
      (int)left.invoke(this, branch.left) +
      (int)right.invoke(this, branch.right);
}

public int add(Leaf leaf) {
  return leaf.value;
}

public int add(Empty empty) {
  return 0;
}

Main.java:
new Add().add(tree)
```

In der Klasse haben wir zusätzlich eine Variable für den Method-Handle Lookup definiert. Über diesen können wir uns die beiden Method-Handles für den linken und rechten Teilbaum des Zweiges erzeugen. Dabei suchen wir jeweils nach der passenden Methode *add* in der Klasse *Add*. Damit hier auf den dynamischen Typen der Parameter geschaut wird, übergeben wir mit *getClass* den dynamischen Typen des linken und rechten Teil des Zweiges. Beim Ausdruck *return* werden die gefundenen Methoden der beiden Method-Handles ausgeführt. Dadurch können wir Multimethoden auch ohne das Visitor Pattern in Java 8 über die neuen Method-Handles simulieren. Nachteile bei dieser Variante: Wir müssen den Rückgabewert der beiden durch den Method-Handle ausgeführten Methoden den Typen umwandeln. Des Weiteren kann unsere Methode nun eine Exception werfen, sofern die Methode nicht gefunden wird. Wenn wir einen der zusammengesetzten Typen (zum Beispiel *add* für das leere Blatt) vergessen, so wird eine Exception geworfen sobald wir einmal den dynamischen Typen *Empty* verwenden. Wir müssen auch beim ersten Aufruf eine dynamische Typ-Umwandlung vornehmen. Außer wir fügen, wie im oberen Code, einen zusätzlichen Method-Handle innerhalb von *add(Tree tree)* hinzu, der die passende Methode für den ersten Aufruf auswählt.

Church Kodierung Grundsätzlich geht es bei der Church Kodierung darum, mit Hilfe von Funktionen höherer Ordnung Datentypen – in unserem Fall algebraische Datentypen – darstellen zu können[KPJ14]. So können wir mit Hilfe des *Function* Interfaces über

Lambda-Ausdrücke Pattern Matching umsetzen. Dazu definieren wir im Interface *Tree* die Methode *match*, welche wir anschließend für das Pattern Matching verwenden können.

```
public <T> T match(Function<Empty, T> a,
                   Function<Leaf, T> b,
                   Function<Branch, T> c);
```

Der Methode übergeben wir für jeden möglichen Typ des Baumes eine Funktion. Als Eingabeparameter erhält die Funktion den jeweiligen Untertypen des Baumes. Der Rückgabewert ist hier generisch über den Typ-Parameter T gebunden. In unserem Beispiel geben wir Integer zurück. Wie beim Visitor Pattern überschreiben wir die Methode in *Empty*, *Leaf* und *Branch*.

```
Überschriebene Methode match in Empty:
return a.apply(this);
Überschriebene Methode match in Leaf:
return b.apply(this);
Überschriebene Methode match in Branch
return c.apply(this);
```

Nun können wir beispielsweise in einer eigenen Klasse die Operationen über Pattern Matching definieren.

```
public int add(Tree t) {
    return t.match(
        (Empty e) -> 0,
        (Leaf l) -> l.value,
        (Branch b) -> b.value + add(b.left) + add(b.right));
}
```

Fertig ist das selbst zusammengebaute destrukturierende Pattern Matching mit Hilfe von Lambda-Ausdrücken in Java 8. Wie beim Visitor Pattern müssen wir eine Methode in jeder Unterklasse des Baumes überschreiben. Dafür kommen wir ohne zusätzliches Visitor Interface und Visitor Implementierungen für die einzelnen Operationen aus.

8.2 Immutability

Generell spielen Immutability und unveränderliche Datenstrukturen in der funktionalen Programmierung eine wichtige Rolle. Dabei geht es darum, dass nach der Erstellung einer Datenstruktur keine Änderungen daran vorgenommen werden können. Sofern sich doch etwas ändert, wird eine neue Instanz der Datenstruktur mit dem geänderten Wert

zurückgegeben. Nicht nur in der funktionalen Programmierung, sondern auch in der objektorientierten Programmierung ergeben sich durch die Verwendung unveränderlicher Objekte Vorteile. Zunächst wollen wir uns in diesem Unterkapitel einige Vor- und Nachteile bei der Verwendung von unveränderlichen Objekten, beziehungsweise Datenstrukturen genauer ansehen. Danach werden wir sehen, in wie fern Unveränderlichkeit in Java 8 umgesetzt werden kann und welche Konstrukte in Java 8 im Sinne der funktionalen Programmierung mit unveränderlichen Datentypen noch nicht in den Standardbibliotheken umgesetzt wurden. Wie wir im Kapitel *Streams API* bereits gesehen haben, unterstützen Streams das Konzept der Unveränderlichkeit. So haben die Operationen nie die darunterliegende Quelle des Streams verändert[GJS+15].

Unveränderliche Objekte sind einfach zu handhaben. Dadurch, dass sie nicht verändert werden können, besitzt diese Art von Datenstruktur oder Objekt genau einen Zustand, welcher sich seit der Erstellung des jeweiligen Objekts nicht verändert hat. Sie können daher als Konstant betrachtet werden[Goo]. Auf Grund dessen kann über Programme, welche größtenteils oder ausschließlich unveränderliche Datenstrukturen verwenden, einfacher argumentiert werden. Solche Programme können dadurch auch einfacher getestet und wiederverwendet werden. Zum anderen sind unveränderliche Objekte oder Datenstrukuren immer von Grund auf *Thread-Safe*. Daher wird kein Synchronisationsmechanismus benötigt und der Einsatz in Hinsicht auf eine parallele Verarbeitung der Daten wird vereinfacht. Durch die inhärente Thread-Sicherheit kann auch zusätzlicher Code eingespart werden[Oraf]. Da die Werte der Objekte oder Datenstrukturen nicht verändert werden können, ist es kein Problem, wenn mehrere Threads auf ein und dieselbe Datenstruktur zugreifen. Somit können unveränderliche Objekte frei geteilt werden. Man muss also beispielsweise nicht für jede Anfrage eine neue Instanz eines Objektes erstellen, sondern kann immer dieselbe Instanz zurückliefern. Dadurch können Ressourcen für die Erstellung der Objekte eingespart werden. Des Weiteren kann auch der Einsatz des Garbage-Collectors der JVM reduziert werden. Unveränderliche Objekte eignen sich auch gut als Konstrukt für andere Objekte. So können diese als Schlüsselwerte für eine *Map* verwendet werden, ohne dass durch eine Veränderung der Werte die Invarianten der Map zunichte gemacht werden.[Blo08]

Es können nicht nur die Objekte an sich ohne Mehraufwand geteilt werden, sondern auch deren innerer Aufbau kann wiederverwendet werden. Das bedeutet, dass bei der Erstellung neuer Objekte auf Inhalte bereits bestehender, unveränderlicher Objekte zurückgegriffen werden kann. Ein gutes Beispiel innerhalb von Java liefert *BigInteger*. Dabei handelt es sich um einen Typ, welcher einen unveränderlichen Integer mit beliebiger Größe darstellt. Dieser wird in Java für arithmetische Berechnungen verwendet. Dabei wird das Vorzeichen des *BigInteger* durch einen Wert vom primitiven Typen *int* dargestellt. Der beliebig große Integer-Wert wird über ein int-Array repräsentiert. Im nachfolgenden Code sehen wir, wie der interne Aufbau für die Erzeugung einer neuen Instanz wiederverwendet werden kann.[Orab]

```
public BigInteger negate() {
    return new BigInteger(this.mag, -this.signum);
}
```

Wenn wir das Vorzeichen eines *BigIntegers* verändern wollen, so müssen wir, da es sich um ein unveränderliches Objekt handelt, eine neue Instanz davon erstellen. Hier wird über die Methode *negate* ein neuer *BigInteger* zurückgegeben. Dabei kann das int-Array, das für die Repräsentation des Wertes zuständig ist, für die Erzeugung des neuen *BigInteger* erneut verwendet werden. Lediglich das Vorzeichen *signum*, wird umgedreht. Ein weiteres Beispiel für unveränderliche Objekte in Java wären Instanzen des Typs *String*[GJS+15].[Blo08]

Ein Nachteil von unveränderlichen Datenstrukturen ist, dass bei der Änderung von Werten immer neue Objekte der jeweiligen Struktur angelegt werden müssen. Dadurch kann es bei häufigen Änderungen und wenig Wiederverwendung des inneren Aufbaus zu Performance- aber auch zu Speicherproblemen kommen.

Nachdem wir nun die Vor- und Nachteile von unveränderlichen Datenstrukturen kennen, wollen wir uns dessen Umsetzung in Java anschauen. In Java können wir über die zwei Schlüsselworte *final* und *private* die Zugriffe und Veränderung von Variablen und Klassen schützen und dadurch Objekte unveränderlich machen. Dazu wird in [Blo08] und [Oraf] beschrieben, wie man eine unveränderliche Klasse erzeugen kann.

- Methoden, die den Zustand des Objekts, also dessen Variablen verändern, sollen vermieden werden. Dementsprechend sollen auch keine *setter*-Methoden erzeugt werden.
- Das Erstellen von Untertypen der Klasse kann durch das Schlüsselwort *final* bei der Definition der Klasse vermieden werden. Eine weitere Möglichkeit wäre es, den Konstruktor als *private* zu deklarieren und Instanzen über Factory-Methoden zu erstellen.
- Alle Variablen der Klasse sollten mit *final* und *private* versehen werden. Durch *final* lässt sich der bei der Erstellung zugewiesene Wert nicht ändern. Durch *private* kann verhindert werden, dass auf Objekte, auf welche durch Variablen der Klasse verwiesen wird, direkt zugegriffen werden kann. Daher kann hier ebenfalls keine Änderung erfolgen.
- Wenn eine Variable einer Klasse auf ein veränderbares Objekt referenziert, so sollten keine direkten Referenzen auf das veränderbare Objekt zurückgeliefert werden. Hier könnte man stattdessen beispielsweise defensive Kopien zurückgeben.
- Wenn über den Konstruktor eine neue Instanz der Klasse erzeugt wird, sollten externe Referenzen auf veränderbare Strukturen nicht direkt gespeichert werden. Auch hier sollten defensive Kopien erzeugt werden.

Der Nachteil, den wir hier bei Java in Hinsicht auf unveränderliche Objekte haben, ist, dass standardmäßig alle Klassen und Variablen diese Veränderung zulassen. Dadurch muss man beim Programmieren darauf achten, vor allem bei Verwendung des funktionalen Programmierparadigmas, die nötigen Modifikatoren korrekt anzuwenden. Dadurch wird die funktionale Programmierung in Java etwas erschwert. Im Gegensatz dazu sind bei Scala alle Datenstrukturen standardmäßig als unveränderlich festgelegt. So auch bei Verwendung von *val* für das Festlegen von Werten[EPFa].

Nun wollen wir noch die Möglichkeiten für unveränderliche Datenstrukturen in Java, wie beispielsweise unveränderliche Listen, näher betrachten. Seit der Einführung des *Java Collection Frameworks* in Java 1.2 im Jahr 1998 gibt es die Möglichkeit, sich für unterschiedliche Kollektionen, wie Listen, Maps und Sets, eine unveränderliche Sicht auf diese über statische Methoden in der Collection API zurückliefern zu lassen. Die dadurch zurückgegebene Liste unterstützt jene Operationen nicht, welche die Liste verändern würden, darunter beispielsweise *add*, *remove* und *addAll*. Beim Aufruf einer dieser Methoden wird eine *UnsupportedOperationException* geworfen. Man muss in diesem Zusammenhang jedoch zwischen zwei Arten der Unveränderlichkeit unterscheiden. Die am Anfang dieses Unterkapitels beschriebene Art der Unveränderlichkeit, welche wir für die funktionale Programmierung anstreben sollten, beschreibt die im Englischen als *Immutability* bezeichnete Eigenschaft von Datenstrukturen. Wobei man hingegen von der Collection API eine Sicht zurückgeliefert bekommt, welche als *Unmodifiable* bezeichnet wird[Staa]. Das folgende Beispiel dazu soll den Unterschied dieser beiden Bezeichnungen verdeutlichen.[Orab]

```
List<Person> list = new ArrayList<>();
list.add(personA);
list.add(personB);

List<Person> unmodifiableList =
    Collections.unmodifiableList(list);

//Ändert den Namen der ersten Person
unmodifiableList.get(0).setName("Name");

//UnsupportedOperationException
unmodifiableList.add(personC);

//personC wird zu list hinzugefügt
//unmodifiableList enthält nun 3 Personen
list.add(personC);
```

Die Liste besteht aus zwei Personen. Über *Collections.unmodifiableList(list)* erhalten wir eine unveränderbare Sicht auf diese Liste. Die Sicht wird auch als *Wrapper*, also als

Verpackung um die eigentliche Liste bezeichnet. Dadurch können über diese Sicht mit dem Namen *unmodifiableList* keine Personen zur Liste hinzugefügt oder entfernt werden, da diese Operationen nicht unterstüzt werden. Jedoch gilt es hier zwei Dinge zu beachten. Zum einen können wir, sofern innerhalb der Klasse Person nicht ausreichend Rücksicht auf Unveränderlichkeit genommen wird, nach wie vor Variablen der Personen, auf die durch die Elemente der Liste referenziert wird, verändern. Das zweite Problem bei der unveränderlichen Sicht auf die Liste ist, dass die darunterliegende Liste nach wie vor von jedem, der eine Referenz auf die eigentliche Liste besitzt, geändert werden kann. *UnmodifiableCollections* besitzen auch einen weiteren Performance-Nachteil. Obwohl sie über den Wrapper nicht verändert werden können, besitzen sie dennoch den gesamten zusätzlichen Code, den auch modifizierbare Datenstrukturen in Java besitzen[Goo]. Wie wir hier sehen, unterstützt Java keine unveränderlichen Datenstrukturen im Sinne von *Immutability*. Dies hat sich auch mit der Veröffentlichung von Java 8 nicht geändert.[Orab]

Wenn man in Java *echte* unveränderliche Datenstrukturen verwenden möchte, so sei hier auf *Google Guava* verwiesen. Dabei handelt es sich um eine Java Bibliothek von Google, welche die unterschiedlichsten funktionalen Konzepte für Java unterstüzt und bereitstellt.[Goo]

8.3 Tail-Recursion

Tail-Recursion, auch Endrekursion genannt, spielt in der funktionalen Programmierung ebenfalls eine wichtige Rolle. In diesem Unterkapitel wollen wir die Probleme von Tail-Recursion betrachten und wie Haskell, Scala, Clojure und Java jeweils damit umgehen können. Funktionale Programmiersprachen besitzen meist keine Schleifen. Um die Funktionsweise von Schleifen dennoch umsetzen zu können, wird stattdessen Rekursion verwendet. Um beispielsweise Iteration von objektorientierten Sprachen zu simulieren wird so genannte Tail-Recursion verwendet. Wie wir bereits wissen, ruft sich bei Rekursion die Funktion innerhalb des Funktionskörpers selbst wieder auf. Bei Tail-Recursion befindet sich dieser weitere Funktionsaufruf ganz am Ende der Funktion. Haskell besitzt eine Art Schleife, welche solange eine Funktion ausführt bis ein bestimmter Wert erreicht wurde. Betrachten wir jedoch die Definition der Funktion *until*, so sehen wir, dass diese auch nur über Rekursion umgesetzt wird.[Jon03]

```
-- Vermeintliche Schleife in Haskell
until :: (a -> Bool) -> (a -> a) -> a -> a

until p f x
    | p x = x
    | otherwise = until p f (f x)
```

Bei den meisten stackbasierten Programmiersprachen wird beim Funktions- beziehungsweise Methodenaufruf ein neuer Stack-Frame erzeugt. Dieser kann erst wieder gelöscht

werden, wenn die jeweilige Funktion vollständig durchlaufen wurde und ein Ergebnis geliefert hat. Dies geschieht bei rekursivem Aufruf jedoch erst dann, wenn die Berechnung vollständig abgearbeitet wurde. Das bedeutet wiederum, je mehr rekursive Aufrufe gemacht wurden, desto mehr Stack-Frames müssen erzeugt werden. Da diese Speicherplatz benötigen, wird ab einer zu hohen Anzahl an rekursiven Methodenaufrufen ein Optimierungsmechanismus benötigt, um Speicherplatz zu sparen und vor einem Überlaufen des Stacks geschützt zu sein. Dieser verwendete Mechanismus wird als *tail call optimization* bezeichnet und ist für die korrekte Funktionsweise von funktionalen Programmiersprachen von Bedeutung. Diese Optimierung muss dabei nicht von der Sprache, sondern von der darunter liegenden Maschine bereitgestellt werden. Im Fall von Java oder anderen JVM-basierenden Sprachen wie Scala oder Clojure wäre dies die JVM.[SO01]

Schauen wir uns zunächst ein Beispiel in Java an. Dabei betrachten wir eine rekursiv aufgebaute Liste von Elementen. Um Tail-Recursion darzustellen, wollen wir die Summe der Listenelemente berechnen. Dazu verwenden wir, wie in einem vorherigen Unterkapitel, eine algebraische Datenstruktur. Auch der im Unterkapitel *Algebraische Datenstrukturen* verwendete Baum könnte hier als Beispiel verwendet werden. Zuerst wird das Beispiel in Haskell umgesetzt.

```
data List a = Element a (List a)
            | Empty

calcSum :: Num a => List a -> a
calcSum (Element x tail) = x + calcSum tail
calcSum (Empty) = 0

--Beispiel liefert 10
calcSum (Element 1 (Element 2 (Element 3 (Element 4 (Empty)))))
```

In Java setzen wir das Beispiel folgendermaßen um. Wir definieren ein Interface für die Liste, implementieren zwei Klassen für dieses Interface. Die eine Klasse Element repräsentiert ein Element der Liste, bestehend aus einem Wert und dem Rest der Liste. Die Klasse Empty repräsentiert ein leeres Listenelement. Wir erzeugen eine Beispielliste folgendermaßen. Die Operationen für die Summe deklarieren wir direkt in den beiden Klassen.

```
Liste liste = new Element(1, new Element(1, new Empty()));
liste.summe();
```

Was passiert nun wenn wir eine Liste mit beispielsweise einhunderttausend Elementen rekursiv über die Methode *summe()* durchlaufen.

```
Exception in thread "main" java.lang.StackOverflowError
at main.Element.summe(Element.java:14)
at main.Element.summe(Element.java:14)
at main.Element.summe(Element.java:14)
at main.Element.summe(Element.java:14)
usw...
```

Wie wir bereits im Kapitel über die JVM gehört haben, arbeitet die JVM ebenfalls Stack-basiert. Jeder Methodenaufruf erzeugt einen neuen Stack-Frame. Bei Schleifen wird nur ein Stack benötigt. Es wird jener Stack, welcher die Schleife durchläuft, für die Berechnungen verwendet. Bei Rekursion hingegen werden die Parameter und sonstige für den Methoden-Aufruf relevante Informationen auf den neuen Stack-Frame geschrieben und bei Beendigung der Methode wird der Rückgabewert ebenfalls auf den Stack gepusht. Nach der Rückgabe kann der jeweilige Speicherplatz für den Frame wieder freigegeben werden. Nach zu vielen rekursiven Aufrufen kommt es zu oben auftretender Fehlermeldung in Java. Es konnte kein neuer Stack-Frame für den Methodenaufruf bereitstellt werden. Das Programm stürzt ab. Die JVM unterstützt nämlich keine Optimierungen in Hinsicht auf tail calls.[Sch09]

Als nächstes betrachten wir wie man tail calls allgemein optimieren kann, wenn die darunterliegende Laufzeit keine tail optimizations unterstützt. Anschließend schauen wir uns die Umsetzung in Scala und Clojure an. Danach werden wir mit Hilfe von Java 8 versuchen einen StackOverflow zu vermeiden.

Tail Call Optimization Wenn man nun die Rekursion so aufbaut, dass der letzte Aufruf dieser Methode der rekursive Aufruf der Methode ist, so gibt es Möglichkeiten die Rekursion zu optimieren. Sofern nämlich der letzte Aufruf der Methodenaufruf selbst ist, benötigen wir den vorherigen Stack nicht mehr. Das bedeutet, wir kehren nicht mehr zur vorherigen Methode zurück. Daher können wir die Rekursion als einfachen Sprung in die neue Methode umsetzen[Jon92]. Wenn tail call optimization so einfach umzusetzen ist, warum wird sie dann dennoch nicht in der JVM unterstützt? Tail call optimization wird dabei nicht unbedingt wegen der besseren Performance benötigt. Es wird von funktionalen Programmiersprachen benötigt, um überhaupt Iteration auf Elemente über Rekursion umsetzen zu können. Der Grund warum die JVM noch immer keine Optimierungen in dieser Hinsicht bietet ist dass die Stack-Frames für Sicherheitsüberprüfungen innerhalb der JVM für die Code-Ausführung benötigt werden. Wenn sich eine rekursive Methode jedoch nur selbst aufruft, ist die sicherheitsrelevante Information, die über den Stack-Frame gewonnen wird, redundant. In [Sch09] wird eine Möglichkeit für tail call optimization in der JVM beschrieben. Dabei werden für sicherheitsrelevante Überprüfungen redundante Stack-Frames für rekursive Aufrufe wiederverwendet. Unterscheiden sich die Stack-Frames jedoch, so wird bei den unterschiedlichen Stack-Frames keine Optimierung durchgeführt, damit die Sicherheit weiterhin gewährleistet wird. Diese Lösung für die JVM wurde aber bisher noch nicht in die offiziellen Releases der *HotSpot JVM* integriert.[Sch09]

Alternativen für Tail Call Optimization Es gibt mehrere Alternativen um Tail-Calls auch ohne offiziellen Support von Tail-Call- Optimization zu optimieren. Diese wollen wir uns anhand von Scala und Clojure ansehen. Da Scala und Clojure ebenfalls auf der JVM laufen und beide Sprachen Tail-Call-Optimization unterstützen, stellt sich nun die Frage inwiefern die beiden Sprachen diese Optimierung umzusetzen versuchen. Da Scala nicht dieselben Sicherheitsmechanismen in Hinsicht auf Stack-Frames benötigt wie Java, kann Tail-Call-Optimization durch so genannte Tail-Call-Elimination relativ einfach über den Compiler umgesetzt werden. Dabei wird im Bytecode die Rekursion durch eine Schleife ersetzt. Dies geschieht auf Bytecode-Level. Der Methodenaufruf wird durch einen *goto* Ausdruck ersetzt[Muc97]. Schauen wir uns nun das Beispiel in Scala an. Listen sind in Scala wie in Haskell rekursiv definiert. Daher können wir direkt unsere Funktion zur Berechnung der Summe schreiben und müssen nicht wie in Java die Klassen für die Liste definieren. Um Tail-Call-Optimization zu unterstützen gibt es in Scala auch eine Annotation für Funktionen *@tailcall*.[EPFa]

```
// nicht Endrekursiv
@tailrec
def sum(ints: List[Int]): Int = ints match {
    case Nil => 0
    case x :: tail => x + sum(tail)
}
```

Die Annotation *@tailrec* hilft uns dabei korrekte Endrekursionen zu schreiben. In diesem Fall beschwert sich der Compiler mit folgender Fehlermeldung: *Recursive call not in tail position.* Obwohl es hier so aussieht, als sei der rekursive Aufruf von *sum(tail)* am Ende der Funktion, ist es nicht so. Tatsächlich muss hier in die ausführende Funktion zurückgekehrt werden, da *x + Ergebnis von sum(tail)* abschließend berechnet wird. Um diese Funktion tail-recursive zu machen müssen wir eine Hilfsfunktion einbauen.

```
def sum(ints: List[Int]): Int = ints {
    @tailrec
    def sumTailRec(ints: List[Int], result: Int): Int = ints match {
        case Nil => result
        case x :: tail => sumTailRec(tail, result + x)
    }

    return sumTailRec(ints, 0);
}
```

Hier verwenden wir eine private Hilfsfunktion für die Berechnung der Summe. Diese Hilfsfunktion ist nun tatsächlich tail-recursive aufgebaut. Dabei übergeben wir jeweils die Zwischenergebnisse der Addition an den nächsten Funktionsaufruf. Dadurch ersparen

wir uns das Zurückspringen in die aufrufende Methode. Hier kann der Scala Compiler die *Tail-Call-Elimination* durchführen. Dadurch erhalten wir bei größeren Listen keinen StackOverflow mehr. Hier der Vergleich der beiden mit javap dekompilierten Varianten.

```
private final int sum_variante1$1(collection.immutable.List);
// Method sumNotRecursive$1:(collection/immutable/List;)I
52: invokespecial #45
```

```
private final int sum_variante2$1(collection.immutable.List, int);
50: astore        8
52: aload         8
54: iload_2
55: iload         7
57: iadd
58: istore_2
59: astore_1
60: goto          0
```

Hier sieht man, dass bei der ersten Variante jeweils der rekursive Aufruf erfolgt. Bei der zweiten Variante werden statt der Rekursion die Werte für die Parameter gesetzt und ein goto repräsentiert eine Ausführung des Codes über eine Schleife, wobei *goto 0* hier an den Funktionsanfang springt.

In Clojure gibt es mehrere Alternativen im Zusammenhang mit Endrekursion und dementsprechender Optimierungen. Einerseits gibt es den Operator *recur* welcher nach Setzen der Parameter an den Funktionsanfang springt. Der Compiler überprüft dabei ähnlich wie in Scala ob dieser Operator auch wirklich am Ende der Funktion steht. Recur funktioniert nur wenn sich die Funktion nur selbst aufruft. Bei abwechselnder Rekursion, also wo sich beispielsweise die Funktionen f und g abwechselnd rekursiv aufrufen, muss die Alternative des Trampolins verwendet werden. Dabei handelt es sich um eine generelle Lösung um StackOverflows durch Rekursion zu vermeiden[TLA92]. Nachteile hierbei sind Performance-Einbußen durch zusätzliche Methodenaufrufe[TBS+15]. In Clojure gibt es in der Standard-Bibliothek die Funktion *trampoline*, welche das Trampolin umsetzt.[Clob]

Beim so genannten Trampolin handelt es sich um eine äußere Funktion, welche wiederholt eine innere Funktion aufruft. Jedes mal wenn die innere Funktion sich über einen Tail Call rekursiv aufrufen möchte, wird stattdessen die innere Funktion an die äußere Funktion (das Trampolin) zurückgegeben. Anschließend ruft das Trampolin die Funktion auf. Die nachfolgende Abbildung zeigt die Funktionsweise eines Trampolins.[SO01]

Lösungsansatz mittels Java 8 Auch in Java 8 können wir nun mit Hilfe eines Trampolins das Problem mit dem überlaufenden Stack umgehen. Dabei erzeugen wir ein funktionales Interface *TailCall*. Die Lösung wurde von Mario Fusco auf der *Lambda World 2015* vorgestellt.[Fus]

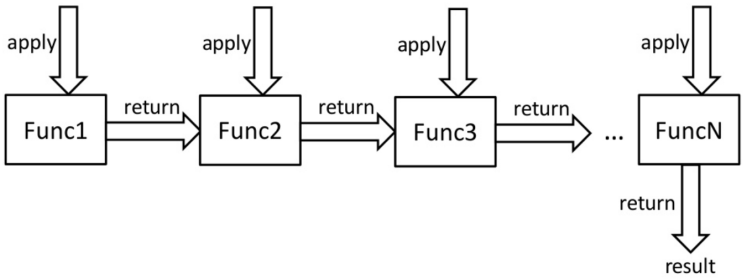


Abbildung 8.2: Funktionsweise Trampolin[Fus]

```
@FunctionalInterface
interface TailCall<T> {
    TailCall<T> apply();
    default boolean isComplete() { return false; }
    default T result() { throw new UnsupportedOperationException(); }
    default T invoke() {
        return Stream.iterate(this, TailCall::apply)
                .filter(TailCall::isComplete)
                .findFirst()
                .get()
                .result();
    }

    static <T> TailCall<T> done(final T value) {
        return new TailCall<T>() {
            @Override
            public TailCall<T> apply() {
                throw new UnsupportedOperationException();
            }

            @Override
            public T result() {
                return value;
            }

            @Override
            public boolean isComplete() {
                return true;
            }
        };
    }
}
```

Dieses Interface mit dem generischen Typen T können wir auch bei anderen Tail Calls verwenden. In der Methode *invoke* können wir über *Stream.iterate* das Trampolin erzeugen. Es ruft so lange rekursiv über das Trampolin die von uns definierte Funktion auf, bis die Methode *isComplete* den Wert *true* zurückliefert. Dort holen wir uns anschließend über *get()* und *result()* das Ergebnis unserer Berechnung. Grundsätzlich unterscheiden wir hier zwischen dem Interface *TailCall* und der Implementierung des Interfaces, welche wir über die statische Methode *done()* zurückbekommen. Das Interface repräsentiert die laufende Berechnung. Die Implementierung die wir von *done* bekommen repräsentiert, dass die Berechnung beendet wurde und wir das Ergebnis abfragen können. Daher ist beim ersten Fall *result* und beim zweiten Fall *apply* nicht implementiert. Um das Trampolin für unser Beispiel verwenden zu können, müssen wir unser Interface und die dazugehörigen Klassen Liste, Element und Empty anpassen.

```
Liste.java:
TailCall<Integer> summe(int summe);

Element.java:
@Override
public TailCall<Integer> summe(int summe) {
    return () -> tail.summe(summe+value);
}

Empty.java:
@Override
public TailCall<Integer> summe(int summe) {
    return TailCall.done(summe);
}
```

Anstatt einen Integer zurückzugeben, wird von den Methoden jeweils ein *TailCall* zurückgegeben. *Element* ist dabei Teil der laufenden Berechnung. Bei *Empty* geben wir *TailCall.done* zurück, zusammen mit dem Ergebnis. Mit *liste.summe(0).invoke()* können wir die rekursive Berechnung über das Trampolin starten. Dadurch erhalten wir keinen StackOverflow mehr. Durch den zusätzlichen Overhead des Trampolins ist die Ausführung jedoch ineffizienter als direkte rekursive Aufrufe.

8.4 Monaden

In diesem Kapitel wollen wir Monaden in der funktionalen Programmierung näher betrachten. Dazu werden wir zuerst über Haskell definieren wie eine Monade aufgebaut ist und welchen Gesetzen sie gehorchen muss, um als Monade zu gelten. Anschließend werden wir überprüfen, ob es in Java 8 Monaden gibt und inwiefern wir diese verwenden können.

Motivation Da es sich bei Haskell um eine pure funktionale Programmiersprache handelt, gibt es keine Seiteneffekte bei der Auswertung von Funktionen. Daher stellt sich nun die Frage, wie man in Haskell seiteneffekt-behaftete Prinzipien der Programmierung, wie beispielsweise globalen Zustand, Exception-Handling, Non-Determinismus oder aber Input-Output-Operationen umsetzen kann, ohne die Eigenschaften von puren Funktionen zu verletzen. Wie wir bereits im Einführungskapitel dieser Arbeit gehört haben, bietet uns die funktionale Programmierung mit puren Funktionen den Vorteil, dass man relativ leicht über Programme schlussfolgern kann. Dadurch entstehen aber auch die folgenden Nachteile. Wenn wir beispielsweise in Haskell Error-Handling hinzufügen wollen, so müssten wir bei einem Programm alle Funktionsaufrufe insofern anpassen, dass bei jedem Aufruf Error-Handling beachtet wird. Ebenso, um bei einer Funktion eine globale Variable hinzuzufügen, müssen alle Funktionsaufrufe um einen zusätzlichen Eingabeparameter erweitert werden. Dieser würde dann die globale Variable von nicht puren oder objektorientierten Programmiersprachen simulieren. Hier kommt nun die Monade ins Spiel. Durch diese kann man sich diesen zusätzlichen Aufwand ersparen und dadurch auch die Modularität erhöhen. Monaden wurden zuerst über die so genannte Kategorientheorie der Mathematik definiert[Str72]. Anschließend wurden Monaden über die Arbeit von Moggi[Mog89] in den Kontext der Programmiersprachen eingeführt.[Wad95]

Definition Die Definition von Monaden werden wir uns zuerst Allgemein und anschließend über Monaden in Haskell ansehen. Monaden bieten uns einen rechnerischen Kontext. Wir übernehmen die Definition der Monade von dem Paper *The essence of functional programming* von Philip Wadler.[Wad92]

Eine Monade ist ein Triple bestehend aus dem Typ-Konstruktor M und einem Paar von Funktionen. Diese Funktionen werden als *unit* und *bind* bezeichnet. Diese Funktionen müssen drei Gesetzen, den so genannten Monaden-Gesetzen, gehorchen.[Wad92]

```
unit :: a -> M a
bind :: M a -> (a -> M b) -> M b
```

M bezeichnet hier den Kontext der Monade. Die Funktion *unit* ermöglicht es uns eine Wert *a* in diesen Kontext einzufügen. Über die Funktion *bind* können wir einen Wert *a* im Kontext von *M* zu einem Wert *b* im selben Kontext umwandeln. Dies geschieht über die übergebene Funktion *(a -> M b)*. Diese nimmt a aus dem Kontext heraus und als Ergebnis erhalten wir den Wert b, welcher wieder im Kontext *M* zu sehen ist. Nachfolgend sehen wir die Definition einer Monade in Haskell[HPF99].

```
class Monad m where
(>>=) :: m a -> (a -> m b) -> m b
(>>) :: m a -> m b -> m b
return :: a -> m a
fail :: String -> m a -- Wird bei Pattern Matching verwendet
```

Die wichtigen Operatoren sind hier (>>=) (bind) und *return* stellt hier die obige Funktion *unit* dar. Der Operator >> ignoriert den Wert *a*, verhält sich ansonsten wie *bind*. Schauen wir uns noch kurz die drei Monaden-Gesetze an, bevor wir uns Monaden in Haskell und deren Verwendung ansehen. Diese müssen erfüllt sein damit es sich tatsächlich um eine Monade handelt. Es reicht also nicht, wenn wir einfach nur den Typ-Konstruktur gemeinsam mit den beiden oben definierten Funktionen verwenden. [HPF99]

- Linke Identität: return a >>= k = k a
- Rechte Identität: m >>= return = m
- Assoziativ: m >>= (\x -> k x >>= h) = (m >>= k) >>= h

Über die linke und rechte Identität sehen wir, dass *return* das neutrale Element ist. Ähnlich wie bei der Addition die Zahl 0. Des Weiteren ist der bind Operator assoziativ. Das bedeutet, dass die Reihenfolge der Auswertung keine Änderung des Ergebnisses hervorruft. Nachfolgend sehen wir die Definition der Monade *Maybe* in Haskell.

```
data Maybe a = Just a | Nothing
```

Die *Maybe-Monade* beschreibt jenen Kontext, in dem ein Wert entweder vorhanden ist oder fehlt. Sie wird hier als algebraischer Datentyp angegeben und besteht aus den beiden Datenstrukturen *Just* und *Nothing*. *Just* repräsentiert, dass ein Typ vom Wert a als Ergebnis vorhanden ist. *Nothing* hingegen repräsentiert den fehlenden Wert. Diesen könnten wir beispielsweise bei der Division verwenden.

```
4 `div` 2 liefer 2
4 `div` 0 liefert Exception
```

Wenn wir die gleichen Beispiele mit / für die Division durchführen, so erhalten wir beim unteren Beispiel als Ergebnis *Infinity*. Wenn wir nun die Division durch Null so darstellen wollen, dass wir kein Ergebnis bekommen, können wir die Maybe Monade einsetzen.

```
divMaybe :: Integer -> Integer -> Maybe Integer
divMaybe  _ 0 = Nothing
divMaybe a b = Just (a `div` b)
```

Hier definieren wir über Pattern Matching *divMaybe*, wobei bei der Division durch Null *Nothing zurückgegeben wird*. Ansonsten wird ganz normal die Division über die bereits vorhandene Funktion *'div'* durchgeführt.

Java 8 Optional Wie wir bereits im Kapitel der Streams gehört haben, wurde mit Java 8 die Klasse *Optional* hinzugefügt. Diese bietet einen Container an, der entweder einen Wert beinhaltet oder die Abwesenheit dieses Wertes repräsentiert. Nachfolgend die zwei wichtigsten Methoden der Klasse *Optional*.[Orab]

```
public final class Optional<T> {
  public static <T> Optional<T> of(T value) {
    return new Optional<>(value);
  }

  // Alternative
  public static <T> Optional<T> ofNullable(T value) {
    return value == null ? empty() : of(value);
  }

  public<U> Optional<U> flatMap(
      Function<? super T, Optional<U>> mapper) {
    Objects.requireNonNull(mapper);
      if (!isPresent())
        return empty();
      else {
        return Objects.requireNonNull(mapper.apply(value));
      }
  }
}
```

Die zwei wichtigsten Methoden sind die statische Methode *Optional.of(T value)* und die Methode *flatMap*. Diese entsprechen den Haskell-Funktionen im Zusammenhang mit Monaden, nämlich *return* und *bind*. Mit *Optional.of* können wir neue Werte in den Kontext packen. Mit Hilfe von *flatMap* können wir eine Funktion auf die Instanz von *Optional* anwenden. Dabei wird der Wert unabhängig vom Kontext der Funktion übergeben. Zurückgegeben wird wieder ein in Optional verpackter Wert. Die Alternative von *Optional.of* ist *Optional.ofNullable*. Das Problem bei der ersten Variante ist, dass wir bei Übergabe von *null* einen Nullpointer erzeugen. Dies kann bei Programmen nützlich sein, wo die Abwesenheit eines Wertes einen Bug im Programm darstellt. Ansonsten sollten wir jedoch *Optional.ofNullable* verwenden. Bei Abwesenheit eines Wertes wird hier eine Singleton-Instanz von Optional namens *EMPTY* zurückgegeben.[Orab]

```
// Statische Variable EMPTY (wie Nothing in Haskell):
private static final Optional<?> EMPTY = new Optional<>();
private Optional() {
    this.value = null;
}
```

Bevor wir mit dem Verwendungszweck von *Optional* in Java 8 fortfahren, schauen wir uns noch die anderen Methoden der Klasse an.

```
public T get()
public boolean isPresent()
public void ifPresent(Consumer<? super T> consumer)
public T orElse(T other)
public T orElseGet(Supplier<? extends T> other)
public <X extends Throwable> T
    orElseThrow(Supplier<? extends X> exceptionSupplier)
// Zusätzlich gibt es noch Funktionen für filter und map
```

Mit *get* können wir den verpackten Wert zurückgeben lassen. Dabei wird jedoch eine Exception geworfen wenn kein Wert vorhanden ist. Daher sollte diese Methoden nur mit Vorsicht aufgerufen werden. Mit Hilfe von *isPresent()* kann abgefragt werden, ob ein Wert vorhanden ist oder nicht. Diese Methoden repräsentiert nur einen Null-Check und erspart uns diesen nicht wirklich. Die restlichen Methoden können verwendet werden um bedingte Aktionen durchzuführen falls der Wert vorhanden ist, oder aber Default-Werte zurückliefern beziehungsweise manuell Exceptions werfen wenn kein Wert vorhanden ist.

Verwendungszweck in Java 8 Mit Hilfe von Optional kann man sich die lästigen Null-Pointer Überprüfungen, die unnötigen Standardcode darstellen, ersparen. Folgendes Beispiel soll dies verdeutlichen. Wir gehen von einer hierarchischen Domain aus: User haben einen Account. Dieser wiederum besitzt Informationen über eine Person. Diese Person hat Kontaktdaten und innerhalb dieser befindet sich unter anderem die E-Mail-Adresse des Users. Diese wollen wir herausfinden. Nachfolgend das Beispiel ohne *Optional*.

```
service.getUser().getPersonalData().getKontaktDaten().getEmail();
```

Wenn ein User keine Personendaten eingetragen hat so erhalten wir hier eine *NullPointerException*. Daher müssen wir das Beispiel mit Java-typischen Null-Checks umsetzen.

```
User user = getUser();
if(user != null) {
  PersonalData personalData = user.getPersonalData();
  if(personalData != null) {
    KontaktDaten kontaktDaten = personalData.getKontaktDaten();
    if(kontaktDaten != null) {
      String email = kontaktDaten.getEmail();
      if(kontaktDaten.getEmail()!= null) {
        // do something
      }}}}
```

Für jeden Zwischenschritt müssen wir einen eigenen Null-Check einbauen um keine Exception zu erhalten wenn ein Wert fehlt. Inwiefern *Optional* überflüssigen Code erspart und den vorhandenen Code lesbarer und wartbarer macht sehen wir im nächsten Code-Block.

```
getUser()
    .flatMap(User::getPersonalData)
    .flatMap(PersonalData::getKontaktDaten)
    .flatMap(KontaktDaten::getEmail)
    .ifPresent(System.out::println);
```

Die einzige Änderung die wir im Code machen müssen, damit wir Optional zusammen mit *flatMap* verwenden können ist, bei den jeweiligen getter-Methoden die einzelnen Objekte in den Kontext von *Optional* zu bringen. Daher verwenden wir zum Beispiel folgenden *Getter* für die Kontaktdaten.

```
public Optional<KontaktDaten> getKontaktDaten() {
    return Optional.ofNullable(kontaktDaten);
}
```

Sofern ein Zwischenwert *null* ist, wird bei unserer Umsetzung oben einfach nichts ausgegeben. Andernfalls erhalten wir die Email-Adresse in der Ausgabe. Mit Hilfe der anderen Methoden in *Optional* können aber auch Default-Aktionen bei Nichtvorhandensein eines Wertes durchgeführt werden.

Die drei Monadengesetze im Zusammenhang mit Optional Damit *Optional* als Monade angesehen werden kann, müssen die drei Monaden-Gesetze erfüllt sein. Überprüfen wir also noch kurz ob diese drei Bedingungen gelten. Dazu verwenden wir die vorher definierten Monaden-Gesetze aus [HPF99]:

```
Linke Identität Definition:
return a >>= k = k a
```

Wenn wir einen Wert a in den monadischen Kontext packen und anschließend die Funktion k anwenden, erhalten wir dasselbe Ergebnis, wie wenn wir die Funktion k auf den Wert a anwenden.

```
Linke Identität Optional:
int a = 2;
Function<Integer, Optional<Integer>> k = x -> Optional.of(x * 2);
Optional.of(a).flatMap(k).equals(k.apply(a));
```

Das erste Gesetz wird durch *Optional* erfüllt. Betrachten wir nun die rechte Identität.

```
Rechte Identität Definition:
m >>= return = m
```

Wenn wir an eine Monade jene Funktion binden, welche uns den ursprünglichen Wert in den monadischen Kontext setzt, so erhalten wir denselben Wert (im monadischen Kontext) zurück.

```
Rechte Identität Optional:
int a = 2;
Optional.of(a).flatMap(Optional::of).equals(Optional.of(a))
```

Als nächstes betrachten wir die Definition des Monaden-Gesetzes der Assoziativität.

```
Assoziativität Definition:
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Einfach ausgedrückt: Wie wir die Klammern setzen beeinflusst nicht unser Ergebnis. Man kann auf die Monade m die Funktion anwenden, welche bei k x 'bind' h herauskommt. Oder man kann zuerst auf m die bind Funktion mit k anwenden und anschließend auf das Ergebnis die Funktion h. Das Endergebnis ist hierbei das Gleiche.

```
Assoziativität Optional:
int a = 2;
Optional<Integer> m = Optional.of(a);
Function<Integer, Optional<Integer>> k =
    x -> Optional.of(x * 2);
Function<Integer, Optional<Integer>> h =
    x -> Optional.of(x * 3);

m.flatMap(x -> k.apply(x).flatMap(h)).equals(
    m.flatMap(h).flatMap(k))
```

Die in Java 8 hinzugekommene Klasse *Optional* bietet also unter anderem die Funktionen um einen Wert in den monadischen Kontext zu verpacken und um Funktionen auf die in den Kontext gepackten Werte anzuwenden. Des Weiteren werden auch die drei monadischen Gesetze erfüllt. Daher ist *Optional* eine Monade.

Optional ist nicht die einzige Monade in Java 8. Auch Streams und *CompletableFuture* bieten Funktionen zum Erstellen und Binden der Monade an und erfüllen die drei monadischen Gesetze. *CompletableFuture* kann für asynchrone Berechnungen verwendet

werden[Orab]. Außerdem können wir für das in Kapitel 4 vorgestellte Problem mit Exceptions innerhalb von Lambda-Ausdrücken ebenfalls Monaden verwenden. Dabei hätten wir einen Wert, welcher das problemlose Durchlaufen eines Streams repräsentiert und einen Wert, welcher einen Fehlerfall, also das Auftreten einer Exception innerhalb des Streams darstellt. Diese Monade müssen wir allerdings selbst programmieren beziehungsweise über eine Bibliothek einbinden.

KAPITEL 9

Related Work

[Goe12a] und [Orad] beschäftigen sich mit der Performance von Lambda-Ausdrücken. In [Goe12a] wurden Performance-Benchmarks von Oracle zwischen Lambda-Ausdrücken und anonymen inneren Klassen durchgeführt. Die Kosten für die Ausführung setzen sich dabei aus *Linkage*, *Capture* und *Invocation* zusammen. Einerseits wird hier die Verlinkung mit und ohne *Capturing* betrachtet. Andererseits wird zwischen der Ausführung mit und ohne Tiered-Compilation unterschieden. *Tiered Compilation* bezeichnet eine Performance-Verbesserung der JVM[Orad]. In allen vier getesteten Varianten werden die Lambda-Ausdrücke insgesamt schneller verlinkt und erfasst. [Kuk13] beschäftigt sich mit den einzelnen Schritten der Verlinkung von Lambda-Ausdrücken. So werden 25% Prozent der Zeit für die Ausführung von *invokedynamic* benötigt, davon wird etwa die Hälfte für die Verlinkung des Method-Handles verbraucht. 44% werden von der LambdaMetaFactory verbraucht, wobei hier die innere Klasse dynamisch erzeugt wird und die CallSite mit dem jeweiligen Method-Handle zurückgeliefert wird. Die dynamische Erstellung der inneren Klasse innerhalb der *LambdaMetaFactory* beansprucht hier die Hälfte der Zeit.

In *Performance of Lambda Expressions in Java 8* [WD15] werden Benchmarks zwischen Streams und Schleifen durchgeführt. Nachfolgend sehen wir die Ergebnisse des Benchmarks:

Experiment	Lambda(ms)	Non-Lambda(ms)	Verbesserung(%)
Zählen von Primzahlen	16.81	16.42	-2.40
Addieren von Zahlen	15.96	16.33	-2.40
Verbinden von Strings	32.78	73.31	55.29
Mapping	70.58	105.80	33.29
Liste filtern	72.91	106.18	33.23

Einen guten Überblick in Hinsicht auf sprachenübergreifende Benchmarks im Zusammenhang mit Lambda-Ausdrücken liefert *Clash of the Lambdas*[BPS14] aus dem Jahr 2014.

Es wurden Java, Scala, C# und F# verglichen. In verschiedenen Experimenten werden Daten über den Einsatz von Streams und Lambda-Ausdrücken verarbeitet. Zusätzlich wurden die beiden Optimierungs-Bibliotheken *ScalaBlitz* und *LinqOptimizer* getestet. Bei den durchgeführten Benchmarks ist Java wesentlich schneller als Scala. Lediglich die optimierten Varianten können bei den umgesetzten Beispielen mit der Performance von Java mithalten.

Das Paper *Performance improvement of using lambda expressions with new features of Java 8 vs. other possible variants of iterating over ArrayList in Java*[Jur18] beschäftigt sich ausgiebig mit einem Performance-Vergleich zwischen Schleifen und Lambda-Ausdrücken. Hier wurden die Testdaten für die Benchmarks mit Hilfe einer Spring Applikation und Hibernate aus einer Datenbank geladen. Anschließend wurden verschiedene Möglichkeiten für die Iteration der erzeugten Collections durchlaufen. Dazu gehören Iteratoren, verschiedene Schleifen und die Streams API. Bei den durchgeführten Benchmarks sind Streams etwas schneller als Schleifen. Hier wurde jedoch kein JMH für die Benchmarks verwendet. Zusätzlich könnte die Spring Applikation die Laufzeit-Performance negativ beeinflussen. Dadurch könnten die Benchmark-Ergebnisse Ungenauigkeiten aufweisen.

In [MKTD17] wurden über zweihundert Open-Source-Projekte auf den Einsatz von Lambda-Ausdrücken innerhalb von Java untersucht. Dabei wurden über einhunderttausend Lambdas und deren Verwendungen analysiert. Hier wird auch geklärt, wie und warum Lambda-Ausdrücke von ProgrammiererInnen eingesetzt werden. Dazu wurden auch die Personen, welche zu den Open Source Projekten beigetragen haben per E-Mail befragt, aus welchen Gründen sie jene Lambda-Ausdrücke verwendet haben. So werden etwa achtundachtzig Prozent der Lambda-Ausdrücke bei Methoden-Aufrufen als Funktionen höherer Ordnung verwendet. Nur vier Prozent der verwendeten Lambda-Ausdrücke werden bei *return* Statements verwendet. Außerdem wurde beispielsweise festgestellt, dass die spezialisierten Funktionen für primitive Typen wie *int* eher selten verwendet werden und daher ein Nachteil in Hinsicht auf die Performance entsteht. Es wurde darüber gesprochen wann EntwicklerInnen selbst neue funktionale Interfaces schreiben und einsetzen. Dies geschieht am häufigsten wenn Exceptions innerhalb der Lambda-Ausdrücke geworfen werden. Als Gründe für die Verwendung von Lambda-Ausdrücken geben die meisten Personen die erhöhte Lesbarkeit des Codes an, beziehungsweise das Verhindern des Erzeugens neuer Klassen.

Die Java 8 Bibliothek *jOOL*[Ede] bietet zusätzliche Stream-Operationen an. Aber auch andere funktionale Aspekte, die in Java 8 zu kurz gekommen sind, werden durch diese Bibliothek zugänglich. So werden Tupel hinzugefügt oder aber ein Interface *Seq*, welches Streams erweitert und zusätzliche Operationen wie bei Scala Collections ermöglicht. Auch das Problem im Zusammenhang mit *CheckedExceptions* wird durch diese Bibliothek, wie in Kapitel 4 besprochen, umgangen.

In *Crossing the Gap from Imperative to Functional Programming through Refactoring*[GFDL13] geht es um das Refactoring von objektorientiertem Code zum funktionalen Pendant. Dabei wurde ein Tool namens *LambdaFicator* vorgestellt. Dieses Tool wird auch in [FGLD13] besprochen. Dadurch werden zwei automatische Umwandlungen ermöglicht.

Zum einen können anonyme innere Klassen in Lambda-Ausdrücke umgewandelt werden. Zum anderen können for Schleifen in Streams transformiert werden. Dazu wird der Daten- und Kontrollfluss des Codes analysiert und etwaige Seiteneffekte müssen geprüft werden.

Das Paper *Clone Refactoring with Lambda Expressions*[TMR17] bespricht die Möglichkeiten, mittels Lambda-Ausdrücken in Java 8 Code-Duplikate zu beseitigen. Des Weiteren wird ein Algorithmus vorgestellt, der die automatische Beseitigung von Code-Duplikaten über Lambdas ermöglicht. Dabei wurden mehrere Open-Source-Projekte auf doppelte Code-Blöcke untersucht. Es geht hier beispielsweise um Methoden, welche vom Verhalten her sehr ähnlich aufgebaut sind. Die Arbeit identifiziert dabei so genannte Typ-2 und Typ-3 Klone, welche bei den untersuchten Projekten vierundneunzig Prozent des doppelten Codes ausmachten. So sind bei Typ-2 Klonen die Methoden bis auf unterschiedliche Namen und Typen der verwendeten Variablen identisch. Bei Typ-3 Klonen gibt es zusätzlich noch Änderungen durch hinzugefügte oder gelöschte Ausdrücke zwischen den duplizierten Methoden.

Changing Engines in Midstream: A Java Stream Computational Model for Big Data Processing[SSG+14] erweitert die in Java 8 hinzugekommene Streams API durch einen so genannten *DistributableStream*. Dieser soll es ermöglichen Stream-Operationen auf verteilten Systemen durchzuführen und deren Ergebnisse anschließend zentral für alle verfügbar zu machen. Bei der Implementierung werden zwei verteilte Frameworks unterstützt, nämlich Apache Hadoop und Oracle Coherence. Für diese werden Methoden bereitgestellt, um Daten aus den beiden Frameworks für einen Stream als Datenquelle zu verwenden und die Ergebnisse über Kollektoren wieder in Apache Hadoop und Oracle Coherence abzuspeichern. Außerdem können DistributableStreams auch lokal eingesetzt werden. In diesem Paper werden mehrere Aufgabenbereiche in Hinsicht auf Streams erwähnt, welche von der herkömmlichen Streams API in Java 8 nicht gelöst werden können. Dazu gehört, dass man beispielsweise verteilte Datenbanken direkt als Quelle für den Stream einsetzen möchte. Ebenso soll es ermöglicht werden, dass entweder ganze Stream-Pipelines auf unterschiedlichen Rechenknoten verarbeitet werden, aber auch einzelne Stream-Operationen sollen auf unterschiedliche Knoten verteilt ausgeführt werden können. Bei der Streams API werden die Ergebnisse nur direkt in Form einer Java Collection abgespeichert. *DistributableStreams* sollen das Abspeichern auf verteilte Collections ermöglichen. Dadurch soll die Rechenleistung mehrerer Computer gebündelt werden, um dadurch eine effizientere Verarbeitung der Streams zu ermöglichen. Der *DistributableStream* wird als eigenständige Bibliothek implementiert, ohne dabei von herkömmlichen Streams zu erben oder umgekehrt. Es werden Methoden zur Umwandlung zwischen Streams und DistributableStreams angeboten. Grundsätzlich funktioniert der verteilte Stream so, dass alle Operationen von einem Client serialisiert werden und dadurch der Stream an die Arbeitsknoten verteilt werden kann. Jeder Knoten verarbeitet dann die gesamte Stream-Pipeline anhand der Daten, die lokal bei jedem Knoten gespeichert wurden. Über eine verteilte Collection werden die Ergebnisse in Form einer globalen Sicht auf das Gesamtergebnis ermöglicht. Die verteilten Collections bieten Ihrerseits wieder die Erzeugung neuer verteilter Streams an. Innerhalb des Papers wurde der verteilte

Stream über zwei Algorithmen getestet. So wurde einerseits ein PageRank-Algorithmus umgesetzt. Dabei geht es darum, die Wichtigkeit von Internetseiten einzustufen. Dies geschieht über Graphen. Andererseits wurde K-Means Clustering umgesetzt. Dieses wird bei Machine Learning und Data Mining zur Gruppierung von Daten verwendet.

Die beiden Papers *Trait-oriented Programming in Java 8*[BMN14] und *Automated Refactoring of Legacy Java Software to Default Methods*[KM17] beschäftigen sich mit den kurz in dieser Arbeit angesprochenen Default-Methoden in Interfaces. Dabei geht es in [BMN14] darum, die default-Methoden in Form von so genannten Merkmalen (Traits) zu verwenden. Traits kommen zum Beispiel in Scala zum Einsatz. Oracle hat Default-Methoden eigentlich nur für die Rückwärtskompatibilität zur Sprache hinzugefügt. Jedoch können wir dadurch auch Traits umsetzen und die Modularität der Software erhöhen. Somit kann man Methoden von mehreren Klassen vererben. Das zweite Paper beschäftigt sich mit der Refaktorisierung von altem Code zu den neuen Default-Methoden. Da es sich dabei nicht um eine triviale Aufgabe handelt wird innerhalb dieses Papers ein effizienter Algorithmus vorgestellt, der diese Umwandlung automatisch unter Einbeziehung eines vorher definierten Regelwerks vornehmen kann.

KAPITEL 10

Conclusion

Java 8 ist mit dem Hinzufügen von Lambda-Ausdrücken, Methoden-Referenzen und der Streams API ein wichtiger Schritt in Richtung funktionaler und paralleler Programmierung gelungen. Anonyme innere Klassen hatten zu viel überflüssigen Code. Lambda-Ausdrücken sind wesentlich lesbarer und unterstützen außerdem Typinferenz über den Kontext der funktionalen Interfaces. Über die nun erlaubten default und statischen Methoden in Interfaces wurde das Hinzufügen von Lambdas und Streams in Hinsicht auf Rückwärts-Kompatibilität unterstützt. Über default-Methoden wurde Mehrfachvererbung hinzugefügt. Da dies keine Mehrfachvererbung von Variablen beinhaltet wird das Diamond-Problem vermieden. Streams bieten mit Hilfe von Lambda-Ausdrücken die Möglichkeit Daten deklarativ zu verarbeiten. Die externe Iteration, die bei Schleifen zum Einsatz kommt, wird hier zu einer internen Iteration. Man gibt nur mehr an was gemacht werden soll. Wie es gemacht wird übernimmt die Streams API. Ein Stream besteht aus einer Quelle, ein bis mehreren Zwischenoperationen und einer terminierenden Operation, wobei die Berechnung auf Grund der verzögerten Auswertung erst durch diese gestartet wird. Dadurch werden unendliche Streams ermöglicht.

Parallele Streams verwenden intern das Fork-Join-Framework, ersparen uns jedoch den zusätzlichen Code, der beim Fork-Join-Framework benötigt wird. Dieses Framework setzt eine Variante eines Teile-und-Herrsche-Algorithmus um. Ein Problem wird dabei rekursiv in kleinere Teile geteilt. Sind diese klein genug, so werden sie direkt gelöst und deren Ergebnisse werden zu einem Gesamtergebnis zusammengefügt. Die Teilprobleme werden als Tasks den Threads zugewiesen. Aus Effizienzgründen wird ein Thread-Pool eingesetzt, welcher Threads wiederverwenden kann. Um bei unausgeglichen komplexen Tasks keine Performance zu verlieren, wendet das Fork-Join-Framework Workstealing zur Lastverteilung an. Zum Durchlaufen der Daten eines Streams wird ein Spliterator verwendet. Dieser wurde speziell für parallele Streams hinzugefügt und funktioniert wie ein Iterator. Zusätzlich kann er Daten des Streams über *trySplit* auf mehrere Tasks aufteilen.

Wir haben auch die Vor- und Nachteile von Streams im Zusammenhang mit Schleifen identifiziert. Streams sind auf Grund der deklarativen Operationen aussagekräftiger als Schleifen. Das Problem lässt sich leichter herauslesen, der Code ist lesbarer und wartbarer. Über die interne Iteration und über vordefinierte Operationen bleiben uns Implementierungsdetails und somit ineffiziente Lösungen erspart, wie wir beispielsweise beim Benchmark von *distinct* gesehen haben. Auch die Unveränderlichkeit der Daten in Hinsicht auf die Streams API ist ein Vorteil. So verändert *sort* nicht mehr die darunterliegende Datenstruktur wie beispielsweise *Collections.sort*. Bei der Gruppierung von Daten haben wir gesehen, dass die Streams API wesentlich lesbarer als Schleifen sind und viel zusätzlichen Code und Implementierungsdetails ersparen. Auch die parallele Verarbeitung der Daten ist leichter möglich. Die Streams API hat jedoch auch Nachteile. Die Verwendung von parallelen Streams ist nicht immer effizient. Streams sind nicht immer lesbarer als Schleifen. Da Streams neu sind, ist die Unterstützung der IDEs noch nicht so ausgereift wie bei den alten Konzepten. Dadurch kann das Debugging erschwert werden. Auch das Exception-Handling im Zusammenhang mit Streams kann zusätzlichen Code bedeuten und die Lesbarkeit negativ beeinflussen. Vor allem checked Exceptions sind hier problematisch und die Lösungsansätze sind entweder umständlich oder umgehen das Typsystem.

Wir haben uns die interne Umsetzung von Lambda-Ausdrücken im Vergleich zu anonymen inneren Klassen über das Tool *javap* und über den Quellcode angesehen. Lambda-Ausdrücke verwenden den neuen JVM-Befehl zum Aufrufen von Methoden namens *invokedynamic*. Dieser wurde wegen der statischen Typisierung der JVM für dynamisch typisierte Programmiersprachen innerhalb der JVM hinzugefügt. Dabei wird die Ausführung der tatsächlich ausgeführten Methode erst zur Laufzeit bestimmt. *Invokedynamic* wird bei der ersten Ausführung der dynamischen Callsite mit der aufzurufenden Methode verbunden. Wenn eine Klasse einen Lambda-Ausdruck beinhaltet, so registriert diese eine Bootstrap-Methode bei der JVM. Diese wird vor dem ersten Aufruf ausgeführt. Sie zeigt auf die Methode *metafactory* der Java-Klasse *LambdaMetafactory* und liefert ein CallSite-Objekt zurück, welches einen Method-Handle beinhaltet. Dieser ist ein Funktionszeiger auf die auszuführende Methode. Der Lambda-Ausdruck wird über die LambdaMetafactory zur Laufzeit über das Bytecode-Tool ASM als synthetische innere Klasse erzeugt. Diese implementiert das jeweilige funktionale Interface und setzt den Funktionskörper des Lambdas als Methode um. Durch die dynamische Erzeugung wird die erzeugte Klasse nicht wie bei anonymen inneren Klassen im Dateisystem angelegt. Durch *invokedynamic* kann die Übersetzungsstrategie in zukünftigen Versionen trotz Rückwärtskompatibilität angepasst werden.

Im Vergleich zu anderen objektorientierten Sprachen haben wir gesehen, dass C-Sharp, C++ und Scala bereits vor Java Lambda-Ausdrücke zur Verfügung gestellt haben. Bei Scala haben wir auch auf lokale Variablen Typinferenz. Daher müssen wir bei Lambda-Ausdrücken bei den Parametern Typen angeben. Dies kann über Funktionstypen gemacht werden. Da die JVM jedoch keine Funktionstypen unterstützt sind diese in Scala nur syntaktischer Zucker und werden vom Compiler in so genannte Typklassen (Traits) umge-

wandelt. Diese sind vergleichbar mit den funktionalen Interfaces bei Lambda-Ausdrücken in Java. Bei Scala sind im Gegensatz zu Java auch nicht finale Variablen innerhalb der Lambda-Ausdrücke erlaubt. Hier geht Java den sichereren Weg und verhindert somit einen Großteil an Seiteneffekten, die bei paralleler Ausführung problematisch werden. Scala beinhaltet keine eigene Streams API. Ein Stream in Scala ist eine sequenzielle Collection von Daten, welche als einfach verkettete Liste mit Memoization umgesetzt wird. Das Pendant zur Streams API in Java stellen hier sequenzielle und parallele Collections in Scala dar. Die Auswertung der Operationen ist im Gegensatz zu Java strikt. Eine verzögerte Auswertung wird bei sequenziellen Collections über Views ermöglicht. Parallele Collections verwenden ebenfalls das Fork-Join-Framework. Anstelle von Spliteratoren werden hier Splitters verwendet, die ähnlich wie in Java agieren. Jedoch kann *trySplit* mehrere Splitter zurückliefern und macht den ursprünglichen Splitter ungültig. Wir haben uns auch den Aufbau der Lambda-Ausdrücke innerhalb der JVM genauer angesehen. Dabei haben wir festgestellt, dass Lambda-Ausdrücke in Scala 2.11 nur syntaktischer Zucker waren. Der Compiler hat, wie bei anonymen inneren Klassen in Java, direkt neue class-Dateien erzeugt. Ab Version 2.12 verwendet Scala ebenfalls *invokedynamic*. Hier wird eine andere Methode der LambdaMetaFactory aufgerufen, welche mehr Kontrolle über die Übersetzung der Lambda-Ausdrücke ermöglicht. Der Vergleich zu C-Sharp und C++ in Hinsicht auf funktionale Programmierung Bezog sich nur auf Lambda-Ausdrücke im Allgemeinen. C-Sharp und C++ erlauben im Gegensatz zu Java ebenfalls nicht finale Variablen innerhalb von Lambda-Ausdrücken. Bei C++ muss zusätzlich angegeben werden, welche Variablen erfasst werden sollen und ob diese per Wert oder Referenz erfasst werden.

Im abschließenden Vergleich mit Konzepten der puren funktionalen Programmiersprache Haskell haben wir festgestellt, dass Java und die JVM trotz Lambda-Ausdrücken und Streams API im Zusammenhang mit funktionaler Programmierung Nachholbedarf haben. So haben wir uns algebraische Datentypen und dazugehöriges Pattern-Matching angesehen. Dabei wollten wir einen Baum als algebraischen Datentypen und Operationen auf diesen umsetzen. Bei algebraischen Datentypen ist es einfach neue Operationen hinzuzufügen und schwer neue Untertypen hinzuzufügen. Bei der objektorientierten Umsetzung über Vererbung ist es genau umgekehrt. Java unterstützt relativ simples Pattern-Matching über die *switch-case* Anweisung. Dabei testet der Compiler nicht auf die Vollständigkeit der Muster. Bei switch-case kann man auch nur einen Teil der vordefinierten Typen verwenden, wie beispielsweise Enums. Enums sind eine Umsetzung von algebraischen Datentypen in Java. Leider konnten wir diese nicht für die Umsetzung unseres Beispiels verwenden, da sie als Singletons umgesetzt sind. Deshalb wollten wir mit Hilfe von überladenen Methoden eine Operationen für die *Addition* der Werte des Baumes festlegen. Dies führte jedoch nicht zum gewünschten Ergebnis, da Java bei Parametern nur auf den statischen Typ achtet und kein mehrfach dynamisches Binden unterstützt. Nur beim Aufrufer wird der dynamische Typ beachtet. Daher verwendet man in objektorientierten Klassen das so genannte Visitor-Pattern. Hier wird durch mehrfach angewandtes, einfaches dynamisches Binden das mehrfach dynamische Binden simuliert. Nachteil ist hier, dass jede Klasse die visit Methode definieren muss und wir ein Interface und jeweils dazugehörigen

Implementierungen pro Operation erstellen müssen. Es lassen sich jedoch einfach neue Operationen hinzufügen. Hier haben wir zwei neue Alternativen mit Java 8 finden können. So kann man einerseits Method-Handles einsetzen um die richtige Methode zur Laufzeit auszuwählen und somit Multimethoden zu simulieren. Nachteil sind hier die Exception, sofern die Methode nicht gefunden wurde und zusätzlich gibt es keine Überprüfung des Compilers ob alle Untertypen des algebraischen Typen berücksichtigt werden. Die zweite Variante war eine neuere Form des Visitor Patterns über Church-Encoding. Ähnlich wie beim Visitor Pattern müssen wir hier einmal eine abstrakte Methode definieren, wobei wir die Anzahl der Untertypen als Parameter, in Form von Funktionen über Lambda-Ausdrücke angeben. Die jeweilige Unterklasse wendet die zugehörige Funktion mittels *apply* an. Die Operationen werden über *match* in Form von selbst zusammengebautem Pattern-Matching umgesetzt.

Danach haben wir die Umsetzung von unveränderlichen Datenstrukturen in Java genauer betrachtet. Diese sind inhärent thread-sicher weil nur lesend zugegriffen werden kann. Dadurch können sie problemlos für parallele Berechnungen verwendet werden, da Seiteneffekte vermieden werden. Java verwendet Immutability bei Strings und BigInteger. Außerdem wird das Konzept auch bei Streams angewandt, weil die Quelle des Streams durch die Operationen nie verändert wird. Außerdem erlauben Lambda-Ausdrücke nur effektiv finale Variablen, was ebenfalls ein Vorteil in Hinsicht auf Thread-Sicherheit und parallele Verarbeitung der Daten ist. Ein Nachteil in Java ist jedoch, dass standardmäßig alle Variablen und Datenstrukturen veränderbar sind. Diese müssen wir mittels *private* und *final* einschränken. Es gibt seit Java 1.2 unveränderliche Listen. Diese sind jedoch nur Sichten auf die veränderlichen Listen. Dabei gibt es mehrere Nachteile. Sobald jemand eine Referenz auf die darunterliegende Liste hat, kann die unveränderbare Liste geändert werden. Bei der Verwendung der nicht unterstützen Methoden add und remove wird erst zur Laufzeit eine Exception geworfen. Die Elemente der Liste sind nicht automatisch unveränderbar. Für echte unveränderliche Datenstrukturen haben wir auf externe Bibliotheken wie Google Guava verwiesen.

Anschließend haben wir uns Funktionen mit Rekursion angesehen. Dabei ruft sich eine Methode selbst auf. Wenn dieser Aufruf am Ende der Funktion ist spricht man von einem Tail-Call. Jeder Funktionsaufruf erzeugt einen neuen Stack-Frame. Bei zu vielen Aufrufen kommt es daher zu einem Stackoverflow. Bei Rekursion wird also ein Optimierungsmechanismus benötigt, für den die JVM zuständig ist. Die JVM unterstützt diese Optimierung jedoch nicht. Bei Tail- Calls benötigen wir den vorherigen Stack nicht mehr. Daher könnte man direkt zum Stack-Frame des ersten Aufrufs zurückspringen. Dies wird in Java auf Grund von Sicherheitsmechanismen nicht ermöglicht. Scala hält sich nicht an die selben Sicherheitsmechanismen, der Compiler ersetzt die Rekursion im Bytecode durch eine Schleife, welche nur einen Stack-Frame benötigen. Clojure löst dieses Problem über ein Trampolin. Dabei wird eine zusätzliche Methode verwendet, welche für die rekursiven Aufrufe zuständig ist. Dadurch kann der Stack-Frame wieder freigegeben werden. In Java können wir dieses Problem mit einem Trampolin, über ein funktionales Interface, mit Hilfe von default-Methoden und Streams elegant lösen.

Abschließend haben wir überprüft ob Java 8 Monaden unterstützt. Diese ermöglichen über die Bereitstellung eines Kontextes die Umsetzung von globalem Zustand, Seiteneffekten durch I/O und Exception-Handling in puren funktionalen Programmiersprachen wie Haskell. Dabei haben wir über die Maybe-Monade in Haskell und die dazugehörigen drei Monaden-Gesetze gesehen, dass auch Java 8 Monaden unterstützt. Diese benötigen einen Typ-Konstruktor der den Kontext darstellt und die Funktionen *unit*, um einen Wert in den jeweiligen Kontext zu setzen, und *bind* zur Anwendung von Funktionen. In Java 8 gibt es über Optional die Möglichkeit, die Abwesenheit von Werten darzustellen. Optional gemeinsam mit den Methoden *Optional.of* und *flatMap* erfüllt die drei Monadengesetze. Futures, welche für asynchrone Berechnungen eingesetzt werden, und Streams sind ebenfalls Monaden.

Einen wichtigen Teil dieser Arbeit stellt das Performance-Kapitel dar. Die Benchmarks wurden mit dem Tool JMH durchgeführt. Dieses führt die einzelnen Benchmarks mehrere tausend Male durch, um so den JIT-Compiler und dadurch Optimierungen von JVM-Bytecode zu Maschinencode zu aktivieren. Es wurden anonyme innere Klassen mit Lambda-Ausdrücken verglichen. Dabei haben wir gesehen, dass diese in etwa gleich schnell ablaufen. Wir haben verschiedene Stream-Operationen und deren Äquivalent mit Schleifen miteinander verglichen. Die Umsetzung mit Schleifen ist meist etwas schneller als jene mit Streams. Davon sind wir auch ausgegangen, da Streams im Hintergrund zusätzlichen Code benötigen. Dadurch, dass Streams ein relativ neues Konzept in Java sind, sind die Ergebnisse dennoch als positiv für Streams einzustufen. Wir dürfen hier nicht die dynamische Übersetzungsstrategie von Lambdas vergessen und dass zukünftige JIT-Compiler bessere Optimierungen für Streams anbieten könnten.

Im Zusammenhang mit parallelen Streams muss man immer den zusätzlichen Aufwand für die Vorbereitung der parallelen Verarbeitung in Betracht ziehen. Einerseits muss die Quelle des Streams für den parallelen Einsatz geeignet sein. Dabei geht es vor allem um die Teilbarkeit der jeweiligen Datenquelle und den synchronisierten Ablauf der Generatoren für Streams. Gute Performance liefern hier Arrays und ArrayListen. Auch beim Zusammenfügen der Ergebnisse spielt die verwendete Collection eine Rolle. Eine *reduce* Operation für die Summe der Elemente ist wesentlich schneller als das Zusammenfügen von Collections zu einem Gesamtergebnis. Hier lohnt sich beispielsweise der Einsatz von ConcurrentMaps. Auch die Anzahl der Tasks und vor allem die Komplexität der Tasks spielt für die parallele Verarbeitung eine wichtige Rolle. Sofern die Tasks sehr rechenintensiv sind rücken die anderen Aspekte für die Verwendung von parallelen Streams in den Hintergrund. Es wurden auch mehrere Fallstricke im Zusammenhang mit Streams vorgestellt. Gerade beim Boxing und Unboxing von Werten muss man beachten, immer den spezialisierten Stream für primitive Datentypen zu verwenden. Außerdem sollte man bei zustandsbehafteten Zwischenoperationen im Zusammenhang mit geordneten parallelen Streams aufpassen. Um diese Ordnung der Elemente über mehrere Threads aufrecht zu erhalten, wird zusätzlicher Speicher und Rechenzeit benötigt. Daher sollte man den Stream, wenn die Reihenfolge keine Rolle spielt als ungeordnet deklarieren und ansonsten sequenzielle Streams verwenden.

Es wurde zusätzlich ein Vergleich zwischen der Hotspot JVM und der J9 durchgeführt. Die Ergebnisse waren hier abhängig vom jeweiligen Benchmark. Bei Schleifen und sequenziellen Streams waren die Ergebnisse eher ausgeglichen. Meist war die Hotspot JVM hier geringfügig schneller. Bei flatMap() und grouping() um etwa dreißig Prozent. Bei der parallelen Verarbeitung war die J9 jedoch sehr überzeugend. Grund dafür ist, dass die J9 keine Referenz-Implementierung der JVM-Spezifikation darstellt und dadurch mehr Freiheiten in Hinsicht auf Optimierungsstrategien hat.

Der abschließende Vergleich zwischen Java und Scala war auf Grund des nicht trivialen Vergleichs zweier Programmiersprachen, mit teilweise unterschiedlichen Implementierungen der Datenstrukturen, etwas komplizierter. Vor allem die strikte Auswertung von Scala im Zusammenhang mit Collections bedeutet hier für Java einen Vorteil in Hinsicht der benötigen Laufzeit. Beim Benchmark haben wir vor allem im Zusammenhang mit Boxing und Unboxing, also bei Verwendung von spezialisierten Streams in Java gesehen, dass Scala hier nicht mithalten kann. Die Performance-Einbuße der strikten Auswertung lässt sich zumindest sequenziell mit Views vermeiden. Bei allen anderen Benchmarks zwischen Java und Scala war das Ergebnis sehr ausgeglichen. Manche Operationen waren in Java schneller, manche in Scala. Teilweise konnte die parallele Verarbeitung in Scala keine nennenswerten Verbesserungen bringen, da der Overhead zu hoch war.

Future Work Es gibt viele Themenbereiche die für zukünftige Arbeiten herangezogen werden können. So können wir gerade im Bereich der dynamischen Übersetzung der Lambda-Ausdrücke in JVM-Bytecode neue Alternativen einsetzen und in diesem Zusammenhang auch weitere Benchmarks durchführen. Auch in Hinsicht auf den Vergleich der Konzepte zwischen Haskell und Java gibt es noch Aufholbedarf. So wurde beispielsweise die verzögerte Auswertung innerhalb der Streams API innerhalb dieser Arbeit nicht genauer analysiert. Auch ein Vergleich zwischen der Streams API und externen funktionalen Bibliotheken wäre sehr interessant. Ebenso gibt es in Hinsicht auf Benchmarks viele Möglichkeiten. So könnte man auch zusätzliche JVM-Implementierungen miteinander vergleichen. Im Vergleich zwischen Java und Scala kann man zusätzliche Performance-Benchmarks durchführen.

Das Hinzufügen von Lambda-Ausdrücken und der Streams API zu Java 8 ist ein wichtiger Schritt um die funktionale Programmierung und die Verarbeitung von Daten auf eine deklarative Weise voranzutreiben. Auch die parallele Verarbeitung von großen Datenmengen wird dadurch einfach ermöglicht. Die dynamische Übersetzungsstrategie ermöglicht hier in zukünftigen Versionen zusätzliche Performance-Verbesserungen, wobei die Streams API schon jetzt gut mit der Laufzeit-Performance von Schleifen mithalten kann. Bei manchen funktionalen Konzepten hat Java noch Nachholbedarf. Hier muss man sich einige Konzepte mühsam umsetzen oder eine externe Bibliothek verwenden. Dazu zählen beispielsweise Pattern-Matching für algebraische Datentypen, Tail-Recursion um Stackoverflows innerhalb von Java zu vermeiden, unveränderliche Datenstrukturen und Monaden um Exception-Handling innerhalb der Streams API einfacher zu realisieren.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AC12] Martin Abadi and Luca Cardelli. *A theory of objects*. Springer Science & Business Media, 2012.

[Ale13] Alvin Alexander. *Scala Cookbook: Recipes for Object-Oriented and Functional Programming*. O'Reilly Media, Inc., 2013.

[Aut] The Project Lombok Authors. Project lombok. `https://projectlombok.org`. (Besucht am 30.06.2016).

[Blo08] Joshua Bloch. *Effective Java (2Nd Edition) (The Java Series)*. Prentice Hall PTR, 2 edition, 2008.

[BMN14] Viviana Bono, Enrico Mensa, and Marco Naddeo. Trait-oriented programming in java 8. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 181–186. ACM, 2014.

[BPS14] Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Clash of the lambdas. *ICOOLPS*, 2014.

[BRA] JET BRAINS. Intellij idea. `https://www.jetbrains.com/idea/`. (Besucht am 22.07.2018).

[BRLG04] Fabian Büttner, Oliver Radfelder, Arne Lindow, and Martin Gogolla. Digging into the visitor pattern. In *SEKE*, pages 135–141, 2004.

[Buc14] Alex Buckley. The road to lambda. `https://www.eclipsecon.org/na2014/sites/default/files/slides/2014-03-18%20The%20Road%20To%20Lambda.pdf`, 2014. (Besucht am 25.02.2016).

[CLCM00] Curtis Clifton, Gary T Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *ACM Sigplan Notices*, volume 35, pages 130–145. ACM, 2000.

[Cloa] Clojure. Clojure. `https://clojure.org/`. (Besucht am 14.02.2016).

[Clob] Clojure. Clojure core api. https://clojure.github.io/clojure/clojure.core-api.html#clojure.core. (Besucht am 22.07.2018).

[Cpp] Cppreference. Cppreference - lambda functions. http://en.cppreference.com/w/cpp/language/lambda. (Besucht am 10.07.2016).

[Ede] Lukas Eder. jool - the missing parts in java 8. https://github.com/jOOQ/jOOL. (Besucht am 31.07.2018).

[EPFa] EPFL. Scala api 2.11.8. http://www.scala-lang.org/api/2.11.8/#scala.Function2. (Besucht am 30.06.2016).

[EPFb] EPFL. Scala documentation. http://docs.scala-lang.org/tutorials/tour/anonymous-function-syntax.html. (Besucht am 30.06.2016).

[FGLD13] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. Lambdaficator: from imperative to functional programming through automated refactoring. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1287–1290. IEEE Press, 2013.

[Fis15] Robert Fischer. *Java Closures and Lambda*. Apress, 2015.

[Fit96] Ronan Fitzpatrick. Software quality: definitions and strategic issues. 1996.

[Fus] Mario Fusco. Laziness, trampolines, monoids and other functional amenities: this is not your father's java. https://de.slideshare.net/mariofusco/lazine. (Besucht am 09.06.2018).

[G+14] Brian Goetz et al. Jsr 335: Lambda expressions for the java programming language, final release. https://jcp.org/aboutJava/communityprocess/final/jsr335/index.html, 2014. (Besucht am 27.02.2016).

[GFDL13] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 543–553. ACM, 2013.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJS+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The java language specification java se 8 edition. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf, Februar 2015. (Besucht am 08.07.2016).

[GL96] John M Gravley and Arun Lakhotia. Identifying enumeration types modeled with symbolic constants. In *Proceedings of the Third Working Conference on Reverse Engineering*, pages 227–236. IEEE, 1996.

[GLS11] Joseph Yossi Gil, Keren Lenz, and Yuval Shimron. A microbenchmark case study and lessons learned. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pages 297–308. ACM, 2011.

[GM95] James Gosling and Henry McGilton. The java language environment. *Sun Microsystems Computer Company*, 2550, 1995.

[Goe12a] Brian Goetz. Lambda: A peek under the hood. *Oracle, JAX London*, 2012.

[Goe12b] Brian Goetz. Translation of lambda expressions. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html`, April 2012. (Besucht am 13.03.2016.

[Goe13a] Brian Goetz. State of the lambda. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html`, September 2013. (Besucht am 25.02.2016).

[Goe13b] Brian Goetz. State of the lambda: Libraries edition. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html`, September 2013. (Besucht am 29.02.2016).

[Gol84] Adele J Goldberg. Smalltalk-80: the interactive programming environment. 1984.

[Goo] Google. Google guava github. `https://github.com/google/guava`. (Besucht am 08.07.2016).

[GS11] Nasser Giacaman and Oliver Sinnen. Parallel iterator for parallelizing object-oriented applications. *International Journal of Parallel Programming*, 39(2):232–269, 2011.

[Has] Haskell. Haskell wiki. `https://wiki.haskell.org/Haskell`. (Besucht am 15.02.2016).

[Hic08] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008.

[HM97] Marty Hall and J Paul McNamee. Improving software performance with automatic memoization. *Johns Hopkins APL Technical Digest*, 18(2):255, 1997.

[HPF99] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to haskell 98, 1999.

[Hug89] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[Hug97] Jim Hugunin. Python and java: The best of both worlds. In *Proceedings of the 6th international Python conference*, volume 9, pages 2–18, 1997.

[Jon92] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, 1992.

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[JPK10] Jan Martin Jansen, Rinus Plasmeijer, and Pieter Koopman. Functional pearl: Comprehensive encoding of data types and algorithms in the *lambda*-calculus. Submitted for publication, 01 2010.

[jru] Jruby.org. `http://jruby.org/`. (Besucht am 20.03.2016).

[Jur18] J Jurinová. Performance improvement of using lambda expressions with new features of java 8 vs. other possible variants of iterating over arraylist in java. *Journal of Applied Mathematics, Statistics and Informatics*, 14(1):103–131, 2018.

[KM17] Raffi Khatchadourian and Hidehiko Masuhara. Automated refactoring of legacy java software to default methods. In *Proceedings of the 39th International Conference on Software Engineering*, pages 82–93. IEEE Press, 2017.

[KPJ14] Pieter Koopman, Rinus Plasmeijer, and Jan Martin Jansen. Church encoding of data types considered harmful for implementations: Functional pearl. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*, page 4. ACM, 2014.

[Kuk13] Sergey Kuksenko. Jdk 8: Lambda performance study. `http://www.oracle.com/technetwork/java/jvmls2013kuksen-2014088.pdf`, 2013. JVM Language Summit.

[Kul07] Eugene Kuleshov. Using the asm framework to implement common java bytecode transformation patterns. *Aspect-Oriented Software Development*, 2007.

[L+14] Doug Lea et al. When to use parallel streams. `http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html`, September 2014. (Besucht am 24.03.2016).

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. *Acm sigplan notices*, 33(10):36–44, 1998.

[Lea00] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM, 2000.

[LYBB15] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The java virtual machine specification. `https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf`, Februar 2015. (Besucht am 11.03.2016).

[M+10] Simon Marlow et al. Haskell 2010 language report. *Available online https://www.haskell.org/definition/haskell2010.pdf*, 2010.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[Mica] Microsoft. C-sharp programming guide. `https://msdn.microsoft.com/en-us/library/ms173171.aspx`. (Besucht am 10.07.2016).

[Micb] Microsoft. Msdn documentation. `https://msdn.microsoft.com/library`. (Besucht am 10.07.2016).

[Micc] Microsoft. .net framework msdn. `https://msdn.microsoft.com/de-de/vstudio/aa496123`. (Besucht am 10.07.2016).

[micd] SUN microsystems. sun.misc: Unsafe.java. `http://www.docjar.com/html/api/sun/misc/Unsafe.java.html`. (Besucht am 15.03.2016).

[MKTD17] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in java. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):85, 2017.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 14–23. IEEE, 1989.

[Muc97] Steven S Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[MVS09] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 45–54, 2009.

[Naf14] Maurice Naftalin. *Mastering Lambdas: Java Programming in a Multicore World*. McGraw Hill Professional, 2014.

[Net07] Sun Developer Network. The java hotspot performance engine architecture. *Sun Microsystem*, 2007.

[Niñ07] Jaime Niño. The cost of erasure in java generics type system. *Journal of Computing Sciences in Colleges*, 22(5):2–11, 2007.

[OAC+04a] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[OAC+04b] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[Ode09] Martin Odersky. Scala 2.8 collections, 2009.

[Oraa] Oracle. Default methods (the java tutorials > learning the java language > interfaces and inheritance). `https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html`. (Besucht am 25.02.2016).

[Orab] Oracle. Java 8 api documentation. `https://docs.oracle.com/javase/8/docs/api/`. (Besucht am 25.02.2016).

[Orac] Oracle. Java compatibility guide. `http://www.oracle.com/technetwork/java/javase/8-compatibility-guide-2156366.html`. (Besucht am 30.06.2016).

[Orad] Oracle. Java hotspot virtual machine performance enhancements. `http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#tieredcompilation`. (Besucht am 22.03.2016).

[Orae] Oracle. Java Programming Language Enhancements. `http://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html`. [Online; Besucht am 24.02.2016].

[Oraf] Oracle. The java tutorials. `https://docs.oracle.com/javase/tutorial/`. (Besucht am 30.06.2016).

[Orag] Oracle. javap - the java class file disassembler. `https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html`. (Besucht am 13.03.2016).

[Orah] Oracle. Openjdk. `http://openjdk.java.net/`. (Besucht am 06.03.2016).

[Orai] Oracle. Openjdk: jmh. `http://openjdk.java.net/projects/code-tools/jmh/`. (Besucht am 22.03.2016).

[Oraj] Oracle. Release notes. https://docs.oracle.com. (Besucht am 01.07.2018).

[Ora04] Oracle. New features and enhancements j2se 5.0. http://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html#forloop, 2004. (Besucht am 29.02.2016).

[ORPS09] Francisco Ortin, Jose Manuel Redondo, and J Baltasar García Perez-Schofield. Efficient virtual machine support of runtime structural reflection. *Science of computer Programming*, 74(10):836–860, 2009.

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–159. ACM, 1997.

[PBRO11] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In *European Conference on Parallel Processing*, pages 136–147. Springer, 2011.

[pip] Pipeline pattern. http://parlab.eecs.berkeley.edu/wiki/_media/patterns/pipeline-v1.pdf. Modified by Yunsup Lee,Ver 1.0 (March 11, 2009), based on the pattern Pipeline Pattern described in section 4.8 of PPP, by Tim Mattson et.al.

[PJ98] Jens Palsberg and C Barry Jay. The essence of the visitor pattern. In *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*, pages 9–15. IEEE, 1998.

[Pon11] Julien Ponge. Fork and join: Java can excel at painless parallel programming too! *Oracle Technology Network, Jul*, 2011.

[Pun07] Franz Puntigam. Skriptum: Objektorientierte programmierung. http://www.complang.tuwien.ac.at/franz/objektorientiert/skript07-1seitig.pdf, 2007.

[RM00] Martin P Robillard and Gail C Murphy. Designing robust java programs with exceptions. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 2–10. ACM, 2000.

[Ros09] John R Rose. Bytecodes meet combinators: invokedynamic on the jvm. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, page 2. ACM, 2009.

[Ros13a] John Rose. Methoddata - hotspot - openjdk wiki. https://wiki.openjdk.java.net/display/HotSpot/MethodData, Juni 2013. (Besucht am 17.03.2016).

[Ros13b] John Rose. Typeprofile - hotspot - openjdk wiki. https://wiki.openjdk.java.net/display/HotSpot/TypeProfile, Mai 2013. (Besucht am 17.03.2016).

[Sch09] Arnold Schwaighofer. Tail call optimization in the java hotspot^{TM} vm. *Master's thesis, Johannes Kepler University Linz*, 2009.

[Ske13] Jon Skeet. *C# in Depth*. Manning Publications Co., Greenwich, CT, USA, 3rd edition, 2013.

[SO01] Michel Schinz and Martin Odersky. Tail call elimination on the java virtual machine. *Electronic Notes in Theoretical Computer Science*, 59(1):158–171, 2001.

[SSG+14] Xueyuan Su, Garret Swart, Brian Goetz, Brian Oliver, and Paul Sandoz. Changing engines in midstream: A java stream computational model for big data processing. *Proceedings of the VLDB Endowment*, 7(13):1343–1354, 2014.

[SSOG93] Jaspal Subhlok, James M Stichnoth, David R O'hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Notices*, volume 28, pages 13–22. ACM, 1993.

[Staa] Why doesn't java 8 include immutable collections? http://programmers.stackexchange.com/questions/221762/why-doesnt-java-8-include-immutable-collections. (Besucht am 08.07.2016).

[Stab] Stackoverflow. How can i throw checked exceptions from inside java 8 streams? https://stackoverflow.com/questions/27644361. (Besucht am 02.08.2018).

[Str72] Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149–168, 1972.

[Str13] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[Sub14] Venkat Subramaniam. *Functional programming in Java: harnessing the power of Java 8 Lambda expressions*. Pragmatic Bookshelf, 2014.

[TBS+15] Tomáš Tauber, Xuan Bi, Zhiyuan Shi, Weixin Zhang, Huang Li, Zhenrui Zhang, and Bruno CDS Oliveira. Memory-efficient tail calls in the jvm with imperative functional objects. In *Asian Symposium on Programming Languages and Systems*, pages 11–28. Springer, 2015.

[TLA92] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling standard ml to c. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(2):161–177, 1992.

[TMR17] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*, pages 60–70. IEEE Press, 2017.

[TOAY13] Yahya Tashtoush, Zeinab Odat, Izzat Alsmadi, and Maryan Yatim. Impact of programming features on code readability. *International Journal of Software Engineering and Its Application*, 7(6):441–458, 2013.

[Tor] Mads Torgersen. New features in c-sharp 7.0. `https://blogs.msdn.microsoft.com/dotnet/2017/03/09/new-features-in-c-7-0/`. (Besucht am 03.08.2018).

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. Java 8 in action. 2014.

[VEM02] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97–106. IEEE, 2002.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992.

[Wad95] Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

[Wam11] Dean Wampler. *Functional Programming for Java Developers: Tools for Better Concurrency, Abstraction, and Agility*. OŔeilly Media, Inc., 2011.

[WD15] A Ward and D Deugo. Performance of lambda expressions in java 8. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, page 119. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015.