



Herbert Prähofer

Funktionale Programmierung in Java

Eine umfassende Einführung

dpunkt.verlag



Herbert Prähofer ist Professor für Informatik an der Johannes Kepler Universität Linz, Österreich. Seine Forschungsschwerpunkte liegen im Bereich der Methoden der Softwareentwicklung und des Software Engineerings, mit aktuellen Schwerpunkten im Bereich statischer Programmanalyse und Produktlinien-Engineering. Er ist Autor und Ko-Autor von über 100 Publikationen in wissenschaftlichen Zeitschriften, Konferenzbänden und Büchern. Seit 2007 ist er im Studium Informatik für die Ausbildung in funktionaler Programmierung zuständig und hält dazu Vorlesungen »Prinzipien von Programmiersprachen«, »Objekt-funktionale Programmierung in Scala« und »Funktionale Programmierung in Java«.

Papier
plus[®]
PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus⁺:

www.dpunkt.plus

Herbert Prähofer

Funktionale Programmierung in Java

Eine umfassende Einführung



dpunkt.verlag

Herbert Prähofer
herbert.praehofer@jku.at

Lektorat: Melanie Feldmann
Copy-Editing: Geesche Kieckbusch, Hamburg
Satz: Ill-satz, www.drei-satz.de
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, www.exclam.de

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN:
Buch 978-3-86490-757-9
PDF 978-3-96088-984-7
ePub 978-3-96088-985-4
mobi 978-3-96088-986-1

1. Auflage 2020
Copyright © 2020 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg



Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger
Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die

Einschweißfolie.

Schreiben Sie uns:

Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: hallo@dpunkt.de.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

*Gewidmet meiner Frau Edith und
meinen Kindern Ulrich und Lea*

Inhaltsverzeichnis

Vorwort

1 Einleitung

- 1.1 Elementare Konzepte und Begriffe
- 1.2 Funktionale Programmierung in Java

2 Sprachliche Grundlagen

- 2.1 Java Generics
 - 2.1.1 Typparameter
 - 2.1.2 Typconstraints
 - 2.1.3 Ko- und Kontravarianz
 - 2.1.4 Typinferenz bei Generics
 - 2.1.5 Schwachstellen der Generics in Java
- 2.2 Default-Methoden
- 2.3 Lambda-Ausdrücke
 - 2.3.1 Formen von Lambda-Ausdrücken
 - 2.3.2 Typ eines Lambda-Ausdrucks
 - 2.3.3 Ausnahmen bei Lambda-Ausdrücken
 - 2.3.4 Closures
- 2.4 Funktionale Interfaces
- 2.5 Methodenreferenzen
- 2.6 Zusammenfassung

3 Programmieren ohne Seiteneffekte

- 3.1 Reine Funktionen

- 3.1.1 Iteration vs. Rekursion
 - 3.1.2 Referentielle Transparenz und Ersetzungsprinzip
 - 3.1.3 Funktionen mit Gedächtnis
- 3.2 Funktionale Ausnahmebehandlung mit Optional
- 3.3 Funktionale Listen
 - 3.3.1 Beispielanwendung
- 3.4 Paare und Tupel
- 3.5 Zusammenfassung

4 Programmieren mit Funktionsparametern

- 4.1 Listenverarbeitung mit Funktionen höherer Ordnung
- 4.2 Flexible Programmschnittstellen
- 4.3 Algorithmen
 - 4.3.1 Tiefensuche
 - 4.3.2 Verallgemeinerung der Suche
- 4.4 Entwurfsmuster
 - 4.4.1 Strategie
 - 4.4.2 Kommando
 - 4.4.3 Besucher
- 4.5 Eingebettete und bedingte Ausführung
 - 4.5.1 Eingebetteter Code
 - 4.5.2 Bedingte Ausführung
 - 4.5.3 Fallunterscheidungen
 - 4.5.4 Typtests
- 4.6 Auswertung nach Bedarf
 - 4.6.1 Faule Iteratoren
 - 4.6.2 Unendliche Folgen
 - 4.6.3 Faule Iteration über die Knoten eines Graphen
- 4.7 Zusammenfassung

5 Kombination von Funktionen

- 5.1 Flüssige Schnittstellen
- 5.2 Funktionskomposition
 - 5.2.1 Aufrufketten beim funktionalen Interface Function
 - 5.2.2 Logische Verknüpfungen bei Predicate
 - 5.2.3 Bilden von Vergleichsketten mit Comparator
 - 5.2.4 Beispiel-Workflows
- 5.3 Kombinator-Parser
 - 5.3.1 Parser und Parser-Ergebnisse
 - 5.3.2 Kombinationsoperatoren
 - 5.3.3 Parser für Boolesche Ausdrücke
- 5.4 Domänen-spezifische Sprachen
 - 5.4.1 Fallbeispiel Zustandsmaschinen
- 5.5 Zusammenfassung

6 Funktoren, Monoide und Monaden

- 6.1 Funktoren
 - 6.1.1 Funktor Optional
 - 6.1.2 Gesetze und Eigenschaften
- 6.2 Monoide und Reduktion
 - 6.2.1 Monoide
 - 6.2.2 Reduktion
 - 6.2.3 Monoide in Java
 - 6.2.4 Reduzierbare Strukturen
 - 6.2.5 Anwendungsbeispiele zur Reduktion mit Monoiden
- 6.3 Monaden
 - 6.3.1 Monade Optional
 - 6.3.2 Monade Parser
 - 6.3.3 Gesetze
 - 6.3.4 Bedeutung von Monaden
 - 6.3.5 MonadPlus: Monade mit monoider Kombination

6.4 Zusammenfassung

7 Streams

7.1 Grundlagen von Streams

- 7.1.1 Ein erstes Beispiel
- 7.1.2 Externe vs. interne Iteration
- 7.1.3 Bedarfsauswertung

7.2 Klassen von Streams

7.3 Stream-Operationen

- 7.3.1 Erzeuger-Operationen
- 7.3.2 Zwischenoperationen
- 7.3.3 Terminal-Operationen

7.4 Collectors

- 7.4.1 Interface Collector
- 7.4.2 Vordefinierte Collectors
- 7.4.3 Downstream Collectors
- 7.4.4 Eine eigene Collector-Implementierung

7.5 Anwendungsbeispiele

- 7.5.1 Ergebnisauswertung mit Streams
- 7.5.2 Wortindex zu einem Text

7.6 Hinweise

- 7.6.1 Einmal-Iteration
- 7.6.2 Begrenzung von unendlichen Streams
- 7.6.3 Zustandslose und zustandsbehaftete Operationen
- 7.6.4 Reihenfolge von Operationen
- 7.6.5 Kombinationen von Operationen

7.7 Interne Implementierung

- 7.7.1 Beispiel

7.8 Zusammenfassung

8 Parallele Streams

8.1 Erzeugen von parallelen Streams

- 8.2 Parallele Ausführung
 - 8.2.1 Spliterators
 - 8.2.2 Ausführung durch Fork/Join-Pool
 - 8.2.3 Konfiguration des Fork/Join-Thread-Pools
- 8.3 Bedingungen bei paralleler Ausführung
 - 8.3.1 Parallele Ausführung und Seiteneffekte
 - 8.3.2 Parallele Ausführung und zustandsbehaftete Berechnungen
 - 8.3.3 Eigenschaften der Parameter von reduce
 - 8.3.4 Paralleles Sammeln
- 8.4 Laufzeit
- 8.5 Zusammenfassung

9 Asynchrone Funktionsketten

- 9.1 Eine Lösung mit parallelen Streams
- 9.2 Asynchrone Lösung mit Futures
- 9.3 CompletableFuture
- 9.4 Asynchrone Programmschnittstellen
- 9.5 CompletableFuture als Promise
- 9.6 Kombination von CompletableFutures
 - 9.6.1 Beispiel
- 9.7 Zusammenfassung

10 Reaktive Streams

- 10.1 Grundlagen
 - 10.1.1 Kontrakt von Observable
 - 10.1.2 Erzeugen von Observables
 - 10.1.3 Anmelden und Abmelden von Observer
- 10.2 Varianten
 - 10.2.1 Single
 - 10.2.2 Completable
 - 10.2.3 Maybe

- 10.3 Hot und Cold Observables
 - 10.3.1 ConnectableObservable
 - 10.3.2 Beispiel Echtzeitdaten
- 10.4 Operationen
 - 10.4.1 Abbildungen
 - 10.4.2 Filtern und Teilmengen
 - 10.4.3 Reduktion
 - 10.4.4 Sammeln
 - 10.4.5 Operationen mit Zeit
 - 10.4.6 Kombinationen
 - 10.4.7 Konvertierungen
 - 10.4.8 Seiteneffekte
- 10.5 Nebenläufigkeit
 - 10.5.1 Serialisierung von nebenläufigen Ereignissen
 - 10.5.2 subscribeOn und Scheduler
 - 10.5.3 observeOn
- 10.6 Fehlerbehandlung
 - 10.6.1 Fehlerereignisse auslösen
 - 10.6.2 Auf Fehler reagieren
- 10.7 Rückstau und Flusskontrolle
 - 10.7.1 Reduktion der Menge der Ereignisse
 - 10.7.2 Flowables
- 10.8 Testen reaktiver Streams
- 10.9 Zusammenfassung

11 Testen mit und von Funktionen

- 11.1 Funktionsparameter bei JUnit 5
- 11.2 AssertJ: Eine DSL für Unit-Tests
- 11.3 Eigenschaftsbasiertes Testen nach QuickCheck
 - 11.3.1 Generatoren von Zufallswerten
 - 11.3.2 Tests
 - 11.3.3 Shrinken der Werte

11.4 Zusammenfassung

12 Weiterführende Konzepte

A Bibliografie

B Laufzeitexperimente Parallele Streams

Index

Vorwort

Dieses Buch gibt eine Einführung in die funktionale Programmierung in der Sprache Java. Wir kennen Java als eine Sprache, die auf dem objektorientierten Programmierparadigma beruht und wir sind gut mit dieser Art der Programmierung vertraut. Und wir wissen, dass die in Java 8 eingeführten Lambda-Ausdrücke die Grundlage für eine funktionale Programmierung bilden. Aber rechtfertigt dieses neue Sprachfeature ein Buch mit 300 Seiten?

Wie wir sehen werden, unterscheidet sich funktionale Programmierung grundsätzlich von unserer gewohnten Welt der imperativen und objektorientierten Programmierung. Schon die Ursprünge sind gänzlich unterschiedlich. Während imperative Programmierung als Abstraktion von Maschinencode und objektorientierte Programmierung aus der Motivation, Dinge der Realität abzubilden, entstanden sind, wurde funktionale Programmierung auf Basis einer mathematischen Theorie, dem Lambda-Kalkül, geschaffen. Mehr erfahren Sie dazu in [Kapitel 1](#) dieses Buches.

Für eine funktionale Programmierung muss man sich daher auf andere Denkweisen, Prinzipien und Programmiermuster einlassen. Dieses Buch will diese Denkweisen, Prinzipien und Programmiermuster mit ihrer Ausgestaltung in Java und die darauf aufbauenden Techniken und Systeme vermitteln. Es orientiert sich dabei

stark an den vielfältigen Ideen, Konzepten und Anwendungen, die die Forschung zur funktionalen Programmierung in 60 Jahren hervorgebracht hat. In dem Sinne kann das Buch auch als ein Versuch gesehen werden, dieses vielfältige Wissen auf eine Programmierung in Java zu übertragen. Das Buch verfolgt daher sehr explizit das Ziel, nicht nur die Sprachkonzepte und die neuen Systeme und Bibliotheken zu beschreiben, sondern ganz besonders auch die grundlegenden Ideen und Prinzipien zu vermitteln und damit ein tieferes Verständnis für die funktionale Programmierung zu schaffen.

Das Buch hat seinen Ursprung in Lehrveranstaltungen zur funktionalen Programmierung, die ich seit mehreren Jahren an der Johannes Kepler Universität Linz halte. Dazu gehört eine Speziallehrveranstaltung zur funktionalen Programmierung in Java und eine Lehrveranstaltung zu den Prinzipien der funktionalen Programmierung, die sich auf die Konzepte der rein-funktionalen Sprache Haskell stützt. Eine weitere wichtige Erfahrung, die in dieses Buch eingeflossen ist, war die Beschäftigung und das Arbeiten mit der Programmiersprache Scala. Scala sehe ich als Vorbild für einen Sprachentwurf, der objektorientierte und funktionale Konzepte systematisch vereint. In diesem Sinne war Scala in vielerlei Hinsicht auch Vorbild für die Gestaltung der funktionalen Konzepte in Java.

Zielgruppe

Das Buch richtet sich an Softwareentwickler bzw. Studenten der Informatik, die bereits Erfahrung mit objektorientierter Programmierung in Java haben und sich in das neue Paradigma der funktionalen Programmierung einarbeiten wollen. Die umfassende Behandlung des Themas sollte den Erwerb von fundierten Kenntnissen der

funktionalen Programmierung in Java ermöglichen. Das Buch kann aber auch verwendet werden, um sich gezielt Kenntnisse von spezifischen Techniken anzueignen. Der folgende Abschnitt »Pfade durch das Buch« informiert über mögliche Kapitefolgen beim Lesen.

Das Buch eignet sich auch besonders als begleitendes Lehrbuch zu Vorlesungen zum Thema funktionale Programmierung in Java. Des Weiteren würde ich das Buch als ergänzendes Lehrbuch zu einer Lehrveranstaltung zu fortgeschrittenen Programmier Techniken in Java oder zu einer sprachunabhängigen Einführung in die funktionale Programmierung empfehlen.

Aufbau des Buchs

Das Buch teilt sich konzeptionell in drei Teile: Im ersten Teil, der die [Kapitel 1](#) und [2](#) umfasst, werden die Grundlagen zu einer funktionalen Programmierung gelegt. Der zweite Teil umfasst die [Kapitel 3](#) bis [6](#) und vermittelt wichtige Prinzipien der funktionalen Programmierung in Java. Im dritten Teil mit den [Kapiteln 7](#) bis [11](#) werden wichtige Techniken, Systeme und Bibliothekskomponenten beschrieben. Das Buch schließt in [Kapitel 12](#) mit einem Blick auf die derzeitigen Grenzen der funktionalen Programmierung in Java und mögliche zukünftige Entwicklungen.

Im Einzelnen behandeln die Kapitel folgende Themen:

- In [Kapitel 1](#) wird eine allgemeine Einführung in die funktionale Programmierung inklusive der geschichtlichen Entwicklung gegeben.
- [Kapitel 2](#) behandelt die sprachlichen Grundlagen von Java für die funktionale Programmierung. Dazu gehören nicht nur die Lambda-Ausdrücke, sondern

auch Generics sowie die neuen Default-Methoden bei Interfaces.

- [Kapitel 3](#) befasst sich mit der Programmierung ohne Seiteneffekte. In diesem Kapitel werden auch elementare funktionale Datenstrukturen eingeführt, die in Folge immer wieder verwendet werden.
- [Kapitel 4](#) geht umfassend auf das Arbeiten mit Funktionsparametern ein. Anhand einer Reihe von Fallbeispielen wird gezeigt, wie mit Funktionsparametern Methoden mit hohem Abstraktionsgrad realisiert werden können.
- [Kapitel 5](#) befasst sich mit der Kombination von Funktionen. Es werden die Prinzipien und Möglichkeiten dargestellt, wie man mit Kombinationsoperatoren aus einfachen funktionalen Bausteinen komplexe Funktionen aufbauen kann.
- Im [Kapitel 6](#) werden die in den vorherigen Kapiteln gefundenen Prinzipien und Programmuster in einen allgemeinen Kontext gestellt. Es wird sich zeigen, dass sich dahinter sehr allgemeine, breit einsetzbare Strukturen verbergen, die ihren Ursprung in der Mathematik haben und damit entsprechende formale Eigenschaften mitbringen.
- [Kapitel 7](#) ist das erste Kapitel zu Techniken und Systemen und enthält eine detaillierte Einführung in das Arbeiten mit den funktionalen Streams.
- In einem eigenen [Kapitel 8](#) gehen wir dann auf die parallele Ausführung bei Streams ein.
- In [Kapitel 9](#) wird die Implementierung asynchroner Tasks auf Basis von CompletableFutures behandelt.
- [Kapitel 10](#) gibt eine detaillierte Einführung in das System RxJava, das eine Implementierung der reaktiven Streams für eine Programmierung von asynchronen Ereignisströmen bereitstellt.
- [Kapitel 11](#) enthält eine kurze Darstellung der Anwendung funktionaler Methoden bei Unit-Tests

sowie eine Einführung in eigenschaftsbasiertes Testen nach der Methode von QuickCheck.

- Im abschließenden [Kapitel 12](#) wird ein Einblick auf weiterführende Konzepte der funktionalen Programmierung gegeben, wie sie in den Sprachen Scala und Kotlin zur Verfügung stehen. Es gibt auch Hinweise auf Features, die bei Java für zukünftige Releases geplant sind.
- Der Appendix beinhaltet die Daten zu den Experimenten, die die Einflüsse unterschiedlicher Faktoren auf die Laufzeit von parallelen Streams zeigen.

Pfade durch das Buch

Abhängig von Interesse und Vorkenntnissen bieten sich mehrere Varianten für die Abfolge der Kapitel dieses Buches an. Auch ein Lesen einzelner Themen ist möglich. In den einzelnen Kapiteln sind des Öfteren Verweise zu Inhalten und Beispielen aus früheren Kapiteln gegeben, die man dann eventuell zielgerichtet nachlesen kann.

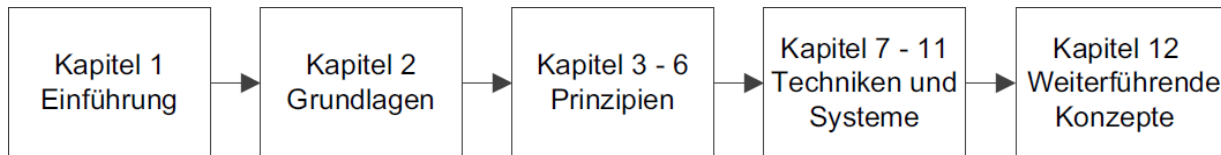
Des Weiteren beinhaltet [Kapitel 5](#) zwei Abschnitte mit fortgeschrittenen Anwendungsbeispielen. Diese Abschnitte können bei der Erstlektüre übersprungen werden, ohne dass das Lesen der folgenden Kapitel beeinträchtigt wird. Auf diese Möglichkeit wird bei den Abschnitten in der Form von Fußnoten hingewiesen.

Im Folgenden sind mehrere mögliche Kapitelfolgen angegeben.

Vollständiger Pfad

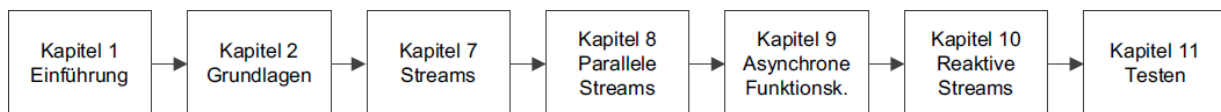
Es ist offenkundig, dass ein Lesen der [Kapitel 1](#) bis [12](#) empfohlen wird, wobei eventuell die gekennzeichneten Abschnitte mit fortgeschrittenen Themen ausgespart werden können. Mit dem Lesen der [Kapitel 3](#) bis [6](#), in

denen die allgemeinen Prinzipien der funktionalen Programmierung vermittelt werden, soll auch ein Verständnis für die Gestaltung der in den [Kapiteln 7 bis 11](#) dargestellten Techniken und Systeme geschaffen werden.



Techniken

Ein weiterer Pfad bietet sich an, wenn man primär an den bekannten Techniken und Systemen und weniger an den allgemeinen Prinzipien der funktionalen Programmierung in Java interessiert ist. Hier besteht die Möglichkeit, mit [Kapitel 1](#) und [2](#) die Grundlagen zu erwerben und dann mit den Techniken und Systemen in den [Kapiteln 7 bis 11](#) fortzufahren. Bei Verweisen zu Inhalten aus früheren Kapiteln kann man diese zielgerichtet nachlesen.

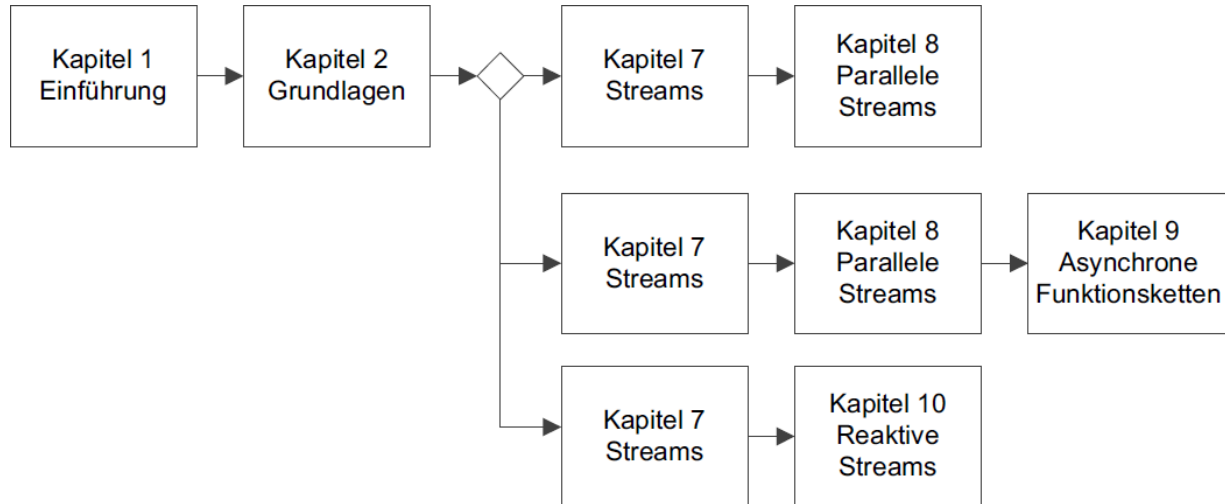


Einzelne Techniken

Das Buch bietet auch die Möglichkeit, sich einzelne Techniken zu erarbeiten. Man startet wieder mit [Kapitel 1](#) und [2](#) zu den Grundlagen und kann dann:

- für Streams mit dem [Kapitel 7](#) gefolgt von [Kapitel 8](#) fortsetzen
- für asynchrone Ausführungsmodelle das [Kapitel 9](#) lesen, wobei hier die parallelen Streams vorausgesetzt werden und daher das Lesen von [Kapitel 7](#) und [8](#) empfohlen wird
- für die reaktiven Streams direkt zum [Kapitel 10](#) gehen, wobei hier entweder Vorkenntnisse zu den

Streams oder das Lesen von [Kapitel 7](#) angeraten ist.



Konventionen

In diesem Buch gelten folgende Konventionen bezüglich Schriftarten und Schreibweisen.

Programme und Programmelemente in Fixpunkt-Font

Für alle Programmbeispiele wird ein Fixpunkt-Font verwendet. Ebenso werden Bezeichner von Programmelementen innerhalb des Fließtextes in Fixpunkt-Font gesetzt. Wird aber der Begriff allgemein verwendet, wird er mit normaler Schriftart geschrieben. Zum Beispiel wird der Begriff String für Zeichenketten mit normaler Schriftart geschrieben, wird aber die Klasse String explizit genannt, wird für den Klassennamen der Fixpunkt-Font verwendet.

Hervorhebungen

Besondere Begriffe werden bei der Einführung *kursiv* geschrieben.

Programmbeispiele

Größere Programmbeispiele, besonders Code von Klassen, werden mit Unterschrift versehen und mit der Kategorie *Listing* nummeriert. Kürzere Anweisungsfolgen werden ohne Nummerierung direkt in den Fließtext eingefügt.

Fachbegriffe und englische Bezeichner

Wenn immer möglich und sinnvoll, werden im Buch deutsche Begriffe verwendet. Bei manchen Begriffen wird aber auf englische Bezeichner zurückgegriffen. So wird zum Beispiel *Map* und nicht *Abbildung* für Programmkomponenten verwendet, die eine Abbildung von Schlüssel auf Werte realisieren. Englische Bezeichner werden dann großgeschrieben, zum Beispiel Streams. Zusammensetzungen aus englischen und deutschen Wörtern werden mit Bindestrich verbunden, zum Beispiel Fixpunkt-Font. Elemente von Programmen werden wie in Java mit Groß- und Kleinschreibung geschrieben, zum Beispiel CompletableFuture.

Anmerkungen und Hinweise

Das Buch gibt an einigen Stellen Zusatzinformationen, Anmerkungen und Hinweise, die über das unmittelbare Thema funktionale Programmierung in Java hinausgehen. Diese sind durch graue Blöcke besonders gekennzeichnet.

Sourcecode zu den Programmbeispielen

Der Sourcecode zu allen Programmbeispielen kann von der Webseite zu diesem Buch

www.dpunkt.de/fpinjava

heruntergeladen werden. Es finden sich hier mehrere Maven-Projekte. Der Code ist in Packages analog zu den

Abschnitten des Buchs organisiert.

Danksagung

Zu diesem Buch haben eine Reihe von Personen beigetragen, bei denen ich mich hier ausdrücklich bedanken möchte.

Ich möchte zuerst meiner Lektorin beim dpunkt.verlag, Frau Melanie Feldmann, für die gründliche und kompetente Lektoratsarbeit danken. Auch möchte ich den anonymen Gutachtern für ihr wertvolles Feedback danken.

Mein besonderer Dank gilt dem Leiter des Instituts für Systemsoftware, Professor Hanspeter Mössenböck, der mir an seinem Institut die Möglichkeit und das Umfeld geboten hat, mich so eingehend mit dem Thema funktionale Programmierung zu beschäftigen. Ich danke ihm auch für das Herstellen des Kontakts zum dpunkt.verlag. Mein Dank geht auch an meine Kollegen am Institut für die immer angenehme und fördernde Arbeitsatmosphäre. Meine Kollegen Kevin Feichtinger und Daniel Hinterreiter waren sehr hilfreich, indem sie erste Versionen von Kapiteln dieses Buches lasen und mir wichtige Rückmeldungen gaben. Meinem Kollegen Professor Paul Grünbacher danke ich für die vielen Jahre der freundschaftlichen Zusammenarbeit und für sein wertvolles Feedback zur Verbesserung der Einleitung zu diesem Buch. Bedanken möchte ich mich auch bei den Kollegen aus dem Oracle-Team, Dr. Roland Schatz und Dr. Lukas Stadler, die mir zu allen Details zur Sprache Java und ihrer Ausführung immer äußerst kompetent helfen konnten.

Meinem ehemaligen Kollegen und Freund Dr. Alexander Fried danke ich ganz besonders für seinen Beitrag zur Gestaltung an und seine Mitarbeit bei unserer gemeinsamen Lehrveranstaltung »Functional Programming

in Java«. Ich danke ihm auch, dass er mich immer ermutigt hat, das Buchprojekt in Angriff zu nehmen.

Schließlich möchte ich mich bei einer Reihe von Studenten bedanken, die durch ihre hervorragenden Studienarbeiten wertvollen Input zu diesem Buch geliefert haben. Namentlich möchte ich hier nennen (in alphabetischer Reihenfolge): Christoph Burghuber, Christoph Gerstberger, Marcel Homolka, Florian Latifi, Daniel Schneider und Florian Schrögendorfer.

Herbert Prähofer

Linz, April 2020

1 Einleitung

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

Funktionale Programmierung hat besonders in den letzten Jahren viel Aufmerksamkeit erhalten. Dabei ist funktionale Programmierung kein neues Paradigma. Tatsächlich ist die erste funktionale Programmiersprache

Lisp fast so alt wie die erste imperative Programmiersprache Fortran. Die Entwicklung der theoretischen Grundlagen zur funktionalen Programmierung, der Lambda Kalkül [17], reicht sogar bis in die 1930er-Jahre zurück. Heute wird funktionale Programmierung von den meisten Programmiersprachen unterstützt.

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

- Mit dem Aufkommen von Prozessoren mit mehreren Kernen werden Softwarekonzepte für eine parallele Ausführung benötigt.
- Mit der zunehmenden Vernetzung, mit Systemen, die ständig online verbunden sind, mit Serverarchitekturen, die zehntausende Klienten gleichzeitig bedienen müssen, und Internet of Things-Anwendungen, die ständig auf Änderungen in der Umwelt reagieren sollen, versagen bisherige Paradigmen der Programmierung.

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

Eigenschaften funktionaler Programme

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

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

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

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

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

Aber ist dieses Modell, mit Funktionen mit Rückgabewerten und mit Funktionsanwendungen zu

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

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

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to

effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

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

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

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

```
int c = 0;

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

Die Lösung baut auf der Wertzuweisung auf. Das Ergebnis wird in der Variablen `c` akkumuliert. In jedem Schleifendurchlauf wird in einer Wertzuweisung ein nächster Wert für `c` berechnet und gespeichert. Die Berechnung ist, wie Backus es kritisiert, »*word-at-a-time style*«.

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

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

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

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

- *Aggregatsfunktionen*: Funktionen, die auf komplexen Datenobjekten als Ganzes operieren und diese ohne