Leseprobe

Mit diesem Buch steigen Sie in Java 9 und die objektorientierte Programmierung ein – ganz ohne Vorkenntnisse. Überzeugen Sie sich in dieser Leseprobe und lassen Sie sich Schritt für Schritt an das erste Java-Programm heranführen. Mit dabei: Übungsaufgaben samt Lösung zur Selbstkontrolle!

Kai Günster

Einführung in Java

734 Seiten, gebunden, 2. Auflage, August 2017
29,90 Euro, ISBN 978-3-8362-4095-6

www.rheinwerk-verlag.de/4096

Kapitel 1
Einführung

Mehr als eine Programmiersprache. Das klingt wie schlechte Fernsehwerbung, und es behauptet jeder von seiner Sprache. Bei Java ist es aber keine Werbeübertreibung, sondern schlicht die Wahrheit, denn zur Plattform Java gehören mehrere Komponenten, von denen die Programmiersprache nur eine ist. In diesem Kapitel lernen Sie die Bausteine kennen, aus denen die Plattform zusammengesetzt ist, sowie eine Entwicklungsumgebung, mit der sie schnell und einfach auf der Plattform Java programmieren.

Warum diese Sprache? Das ist wohl die Frage, die jede Einführung in eine Programmiersprache so schnell wie möglich beantworten sollte. Schließlich wollen Sie, lieber Leser, wissen, ob Sie im Begriff sind, auf das richtige Pferd zu setzen.

Also, warum Java? Programmiersprachen gibt es viele. Die Liste von Programmiersprachen in der englischsprachigen Wikipedia (*http://en.wikipedia.org/wiki/List_of_programming_languages*) hat im Juli 2017 mehr als 650 Einträge. Selbst wenn 90 % davon keine nennenswerte Verbreitung (mehr) haben, bleiben mehr Sprachen übrig, als man im Leben lernen möchte, manche davon mit einer längeren Geschichte als Java, andere jünger und angeblich hipper als Java. Warum also Java?

Java ist eine der meistgenutzten Programmiersprachen weltweit (Quelle: *http://www.tiobe.com/tiobe-index*, Juni 2017). Zwar ist »weil es schon so viele benutzen« nicht immer ein gutes Argument, aber im Fall von Java gibt es sehr gute Gründe für diese Popularität. Viele davon sind natürlich Vorteile der Sprache Java selbst und der Java-Plattform – dazu gleich mehr –, aber der vielleicht wichtigste Grund ist ein externer: Java hat eine unerreichte Breite von Anwendungsgebieten. Es kommt in kritischen **Geschäftsanwendungen** ebenso zum Einsatz wie in Googles mobilem Betriebssystem **Android**. Java ist eine beliebte Sprache für **serverseitige Programmierung im World Wide Web** wie auch für Desktopanwendungen. In Java geschriebene Programme laufen auf fast jedem Computer, egal, ob dieser mit Linux, Windows oder Mac OS betrieben wird – und im Gegensatz zu anderen Programmiersprachen gibt es dabei nur eine Programmversion, nicht eine für jedes unterstützte System, denn Java-Programme sind **plattformunabhängig**.

Neben dem breiten Einsatzgebiet hat die Sprache Java auch, wie oben bereits erwähnt, innere Werte, die überzeugen. Oder besser: Die Plattform Java hat auch innere Werte, denn zu Java gehört mehr als nur die Sprache.

1.1 Was ist Java?

Die Programmiersprache Java, um die es in diesem Buch hauptsächlich gehen soll, ist nur ein Bestandteil der Java-Plattform. Zur Plattform gehören neben der Sprache die *Laufzeitumgebung*, das Programm, das Java-Programme ausführt, und eine umfangreiche *Klassenbibliothek*, die in jeder Java-Installation ohne weiteren Installationsaufwand zur Verfügung steht. Betrachten wir die einzelnen Teile der Java-Plattform etwas näher.

1.1.1 Java – die Sprache

Das Hauptmerkmal der Programmiersprache Java ist, dass sie *objektorientiert* ist. Das ist heute nichts Besonderes mehr, fast alle neueren Programmiersprachen folgen diesem Paradigma. Aber als Java entstand, im letzten Jahrtausend, fand gerade der Umbruch von prozeduraler zu objektorientierter Programmierung statt. Objektorientierung bedeutet in wenigen Worten, dass zusammengehörige Daten und Operationen in einer *Datenstruktur* zusammengefasst werden. Objektorientierung werde ich in Kapitel 5, »Klassen und Objekte«, und Kapitel 6, »Objektorientierung«, eingehend beleuchten.

Viele Details der Sprache hat Java vom prozeduralen **C** und dessen objektorientierter Erweiterung **C++** geerbt. Es wurde aber für Java vieles vereinfacht, und häufige Fehlerquellen wurden entschärft – allen voran das Speichermanagement, für das in C und C++ der Entwickler selbst verantwortlich ist. Das Resultat ist eine Sprache mit einer sehr einfachen und einheitlichen Syntax ohne Ausnahmen und Sonderfälle. Dieser Mangel an »syntaktischem Zucker« wird gerne kritisiert, weil Java-Code dadurch eher etwas länger ist als Code in anderen Sprachen. Andererseits ist es dadurch einfacher, Java-Code zu lesen, da man im Vergleich zu diesen Sprachen nur eine kleine Menge an Sprachkonstrukten kennen muss, um ihn zu verstehen.

Aber Java ist eine lebendige Sprache, die mit neuen Versionen gezielt weiterentwickelt wird, um für bestimmte Fälle ebendiesen Zucker doch zu bieten. In Version 8 von Java hielt eine Erweiterung Einzug in die Sprache, nach der die Entwickler-Community seit Jahren gefleht hatte: Lambda-Ausdrücke (siehe Kapitel 11). Dabei handelt es sich um ein neues Sprachkonstrukt, das die syntaktische Einfachheit der Sprache nicht zerstört, sondern an vielen Stellen klareren, verständlicheren Code möglich macht. Ihre Einführung sind ein gutes Beispiel dafür, dass die Entwickler der Sprache

Java auf die Wünsche ihrer Community eingehen, dabei aber keine überstürzten Entscheidungen treffen – eine gute Kombination für langfristige Stabilität.

1.1.2 Java – die Laufzeitumgebung

Traditionell fallen Programmiersprachen in eine von zwei Gruppen: *kompilierte* und *interpretierte Sprachen*. Bei kompilierten Sprachen, zum Beispiel **C**, wird der Programmcode einmal von einem *Compiler* in Maschinencode umgewandelt. Dieser ist dann ohne weitere Werkzeuge ausführbar, aber nur auf einem Computer mit der Architektur und dem Betriebssystem, für die das Programm kompiliert wurde.

Im Gegensatz dazu benötigen interpretierte Sprachen wie **Perl** oder **Ruby** ein zusätzliches Programm, den *Interpreter*, um sie auszuführen. Die Übersetzung des Programmcodes in Maschinencode findet erst genau in dem Moment statt, in dem das Programm ausgeführt wird. Wer ein solches Programm nutzen möchte, muss den passenden Interpreter auf seinem Computer installieren. Dadurch sind Programme in interpretierten Sprachen auf jedem System ausführbar, für das der Interpreter vorhanden ist. Interpretierte Sprachen stehen aber im Ruf, langsamer zu sein als kompilierte Sprachen, da das Programm zur Laufzeit noch vom Interpreter analysiert und in Maschinencode umgesetzt werden muss.

Java beschreitet einen Mittelweg zwischen den beiden Ansätzen: Java-Code wird mit dem Java-Compiler `javac` kompiliert, dessen Ausgabe ist aber nicht Maschinencode, sondern ein Zwischenformat, der *Java-Bytecode*. Man braucht, um ein Java-Programm auszuführen, immer noch einen Interpreter, aber die Umsetzung des Bytecodes in Maschinencode ist weniger aufwendig als die Analyse des für Menschen lesbaren Programmcodes. Der Performance-Nachteil wird dadurch abgeschwächt. Dennoch stehen Java-Programme manchmal in dem Ruf, sie seien langsamer als Programme in anderen Sprachen, die zu Maschinencode kompiliert werden. Dieser Ruf ist aber nicht mehr gerechtfertigt, ein Performance-Unterschied ist im Allgemeinen nicht feststellbar.

Der Interpreter, oft auch als Java Virtual Machine (JVM) bezeichnet, ist ein Bestandteil der Java *Laufzeitumgebung* (Java Runtime Environment, JRE), aber die Laufzeitumgebung kann mehr. Sie übernimmt auch das gesamte Speichermanagement eines Java-Programms. C++-Entwickler müssen sich selbst darum kümmern, für alle ihre Objekte Speicher zu reservieren, und vor allem auch darum, ihn später wieder freizugeben. Fehler dabei führen zu sogenannten Speicherlecks, also zu Fehlern, bei denen Speicher zwar immer wieder reserviert, aber nicht freigegeben wird, so dass das Programm irgendwann mangels Speicher abstürzt. In Java muss man sich darüber kaum Gedanken machen, die Speicherfreigabe übernimmt der *Garbage Collector*. Er erkennt Objekte, die vom Programm nicht mehr benötigt werden, und gibt ihren Speicher automatisch frei – Speicherlecks sind dadurch zwar nicht komplett ausgeschlos-

sen, aber fast. Details zum Garbage Collector finden Sie in Kapitel 18, »Hinter den Kulissen«.

Zur Laufzeitumgebung gehört außerdem der Java-Compiler HotSpot, der kritische Teile des Bytecodes zur Laufzeit in »echten« Maschinencode kompiliert und Java-Programmen so noch einen Performance-Schub gibt.

Implementierungen des Java Runtime Environments gibt es für alle verbreiteten Betriebssysteme, manchmal sogar mehrere. *Die* Implementierung der Laufzeitumgebung ist natürlich die von Oracle (ursprünglich Sun Microsystems, die aber im Jahre 2010 von Oracle übernommen wurden), aber daneben gibt es weitere, zum Beispiel von IBM, als Open-Source-Projekt (OpenJDK) oder bis Java 6 von Apple. Da das Verhalten der Laufzeitumgebung streng standardisiert ist, lassen sich Programme fast immer problemlos mit einer beliebigen Laufzeitumgebung ausführen. Es gibt zwar Ausnahmefälle, in denen sich Programme in verschiedenen Laufzeitumgebungen unterschiedlich verhalten, aber sie sind erfreulich selten.

Android

Die von Android verwendete ART-VM steht (wie ihr Vorgänger Dalvik) abseits der verschiedenen JRE-Implementierungen: Die verwendete Sprache ist zwar Java, aber der Aufbau der Laufzeitumgebung ist von Grund auf anders, und auch der vom Compiler erzeugte Bytecode ist ein anderer. Auch die Klassenbibliothek (siehe nächster Abschnitt) von ART unterscheidet sich von der Standardumgebung.

Die Java-Laufzeitumgebung genießt selbst unter Kritikern der Sprache einen sehr guten Ruf. Sie bietet ein zuverlässiges Speichermanagement, gute Performance und macht es so gut wie unmöglich, den Rechner durch Programmfehler zum Absturz zu bringen. Deshalb gibt es inzwischen eine Reihe weiterer Sprachen, die mit der Programmiersprache Java manchmal mehr, aber häufig eher weniger verwandt sind, die aber auch in Java-Bytecode kompiliert und vom JRE ausgeführt werden. Dazu gehören zum Beispiel Scala, Groovy und JRuby. Diese Sprachen werden zwar in diesem Buch nicht weiter thematisiert, aber es handelt sich um vollwertige, ausgereifte Programmiersprachen und nicht etwa um »Bürger zweiter Klasse« auf der JVM. Die Existenz und Popularität dieser Sprachen wird allgemein als gutes Omen für die Zukunft der Java-Plattform angesehen.

1.1.3 Java – die Standardbibliothek

Der dritte Pfeiler der Java-Plattform ist die umfangreiche Klassenbibliothek, die in jeder Java-Installation verfügbar ist. Enthalten sind eine Vielzahl von Werkzeugen für Anforderungen, die im Programmieralltag immer wieder gelöst werden müssen: mathematische Berechnungen, Arbeiten mit Zeit- und Datumsangaben, Dateien

lesen und schreiben, über ein Netzwerk kommunizieren, kryptografische Verfahren, grafische Benutzeroberflächen ... Das ist nur eine kleine Auswahl von Lösungen, die die Java-Plattform schon von Haus aus mitbringt. Die Klassenbibliothek ist auch der Hauptunterschied zwischen den verschiedenen verfügbaren Java-Editionen:

- Die *Standard Edition (Java SE)*: Wie der Name schon sagt, ist dies die Edition, die auf den meisten Heimcomputern installiert ist. Die Klassenbibliothek enthält alles oben Genannte. Dieses Buch befasst sich, einige Ausblicke und Anmerkungen ausgenommen, mit der Standard Edition in der aktuellen Version 9. Die gesamte Klassenbibliothek ist Open Source, der Quellcode ist einsehbar und liegt einer Installation des Java Development Kits (JDK) bei.
- Die *Micro Edition (Java ME)*: Diese schlanke Java-Variante ist für Mobiltelefone ausgelegt, deren Hardware hinter der von aktuellen Smartphones mit iOS oder Android zurückbleibt, also ältere Telefone oder solche aus dem Niedrigpreissegment. Die Micro Edition ist auf die Sprachversion 1.3 aus dem Jahr 2000 eingefroren und bietet keine der neueren Sprachfeatures. Die Klassenbibliothek befindet sich auf einem ähnlichen Stand, enthält aber nicht alle Klassen und Funktionen, die in der Standard Edition 1.3 verfügbar waren. Java ME hat durch die aktuellen, leistungsstarken Handys stark an Bedeutung verloren und entwickelt sich nur noch schleppend bis gar nicht weiter.
- Die *Enterprise Edition (Java EE)*: Die Enterprise Edition erweitert die Standard Edition um viele Features für Geschäftsanwendungen, zum Beispiel für die Kommunikation in verteilten Systemen, für transaktionssichere Software oder für das Arbeiten mit Objekten in relationalen Datenbanken. In Kapitel 14, »Servlets – Java im Web«, werden wir uns mit dem Webanteil der Enterprise Edition, der *Java-Servlet-API*, beschäftigen.

1.1.4 Java – die Community

Neben den genannten Bestandteilen der Java-Plattform gibt es einen weiteren wichtigen Pfeiler, der zum anhaltenden Erfolg von Java beiträgt: die Community. Java hat eine sehr lebendige und hilfsbereite Entwicklergemeinschaft und ein riesiges Ökosystem an Open-Source-Projekten. Für fast alles, was von der Standardbibliothek nicht abgedeckt wird, gibt es ein Open-Source-Projekt (häufig sogar mehrere), das die Lücke füllt. In Java ist es häufig nicht das Problem, eine Open-Source-Bibliothek zu finden, die ein Problem löst, sondern eher aus den vorhandenen Möglichkeiten die passende auszuwählen.

Aber die Community leistet nicht nur im Bereich Open-Source-Software sehr viel, sie ist auch durch den *Java Community Process* (JCP) an der Entwicklung der Java-Plattform selbst beteiligt. Unter *https://www.jcp.org/en/home/index* kann jeder kostenlos Mitglied des JCPs werden (Unternehmen müssen einen Mitgliedsbeitrag zahlen) und

dann Erweiterungen an der Sprache vorschlagen oder darüber diskutieren. Selbst wenn Sie nicht mitreden möchten, lohnt sich ein Blick in die offenen JSRs (*Java Specification Requests*), um sich über die Zukunft von Java zu informieren.

1.1.5 Die Geschichte von Java

Java hat es weit gebracht für eine Sprache, die 1991 für interaktive Fernsehprogramme erfunden wurde. Das damals von James Gosling, Mike Sheridan und Patrick Naughton unter dem Namen **Oak** gestartete Projekt erwies sich zwar für das Kabelfernsehen dieser Zeit als zu fortschrittlich, aber irgendwie wurde die Sprache ja trotzdem erfolgreich. Es sollten allerdings noch vier Jahre und eine Namensänderung ins Land gehen, bevor 1996 Java 1.0.2 veröffentlich wurde: Oak wurde in Java umbenannt, das ist US-Slang für Kaffee, den die Entwickler von Java wohl in großen Mengen vernichteten.

Die Geschichte von Java ist seitdem im Wesentlichen einfach und linear, es gibt nur einen Strang von Weiterentwicklungen. Etwas verwirrend wirkt die Versionsgeschichte nur dadurch, dass sich das Schema, nach dem Java-Versionen benannt werden, mehrmals geändert hat. Tabelle 1.1 gibt einen Überblick.

Version	Jahr	Beschreibung
JDK 1.0.2 Java 1	1996	die erste stabile Version der Java-Plattform
JDK 1.1	1997	wichtige Änderungen: *inner classes* (siehe Kapitel 6, »Objektorientierung«) und *Reflection* (siehe Kapitel 4, »Wiederholungen«), Anbindung von Datenbanken durch *JDBC*
JDK 1.2 J2SE 1.2	1998	Die erste Änderung der Versionsnummerierung, JDK 1.2 wurde später in J2SE umbenannt, für Java 2 Standard Edition. Mit dem Sprung auf Version 2 wollte Sun den großen Fortschritt in der Sprache betonen, die Standard Edition wurde hervorgehoben, weil J2ME und J2EE ihrer eigenen Versionierung folgten. Wichtige Änderungen: Das *Swing-Framework* für grafische Oberflächen wurde Teil der Standard Edition, ebenso das *Collections-Framework* (siehe Kapitel 10, »Arrays und Collections«).
J2SE 1.3	2000	J2SE 1.3 war die erste Java-Version, die mit HotSpot ausgeliefert wurde.

Tabelle 1.1 Java-Versionen und wichtige Änderungen im Kurzüberblick

Version	Jahr	Beschreibung
J2SE 1.4	2002	J2SE 1.4 führte das `assert`-Schlüsselwort ein (siehe Kapitel 9, »Fehler und Ausnahmen«). Außerdem enthielt diese Version weitreichende Erweiterungen an der Klassenbibliothek, unter anderem Unterstützung für reguläre Ausdrücke (siehe Kapitel 8, »Die Standardbibliothek«) und eine neue, performantere I/O-Bibliothek (siehe Kapitel 12, »Dateien, Streams und Reader«).
J2SE 1.5 J2SE 5.0	2004	In dieser Version wurde das Schema für Versionsnummern erneut geändert, J2SE 1.5 und J2SE 5.0 bezeichnen dieselbe Version. Diese Version enthielt umfangreiche Spracherweiterungen: ▸ *Generics* für typsichere Listen (Kapitel 10, »Arrays und Collections«) ▸ *Annotationen* (siehe Kapitel 6, »Objektorientierung«) ▸ *Enumerations* (Typen mit aufgezählten Werten, siehe Abschnitt 6.5) ▸ *Varargs* (variable Parameterlisten, siehe Kapitel 10, »Arrays und Collections«) ▸ und mehr Die Klassenbibliothek wurde durch neue Werkzeuge für Multithreading (siehe Kapitel 13) erweitert.
Java SE 6	2006	Java SE 6 führte das heute noch verwendete Versionsschema ein. Änderungen an der Plattform waren weniger umfangreich als in den beiden vorherigen Versionen und betrafen fortgeschrittene Features, die in dieser Einführung zu sehr in die Tiefe gehen würden.
Java SE 7	2011	Trotz der langen Zeit zwischen Java SE 6 und 7 sind die Änderungen an der Sprache Java weniger umfangreich als in vorigen Versionen. Hauptsächlich zu nennen ist die neue, vereinfachte Syntax, um Ressourcen, wie zum Beispiel Dateien, nach Gebrauch zuverlässig zu schließen (das Konstrukt *try-with-resources* in Kapitel 9, »Fehler und Ausnahmen«). Unter der Haube wurde der Bytecode um eine neue Anweisung (`invokedynamic`) erweitert, die die JVM für bestimmte andere Sprachen zugänglicher macht. Treibende Kraft war hierbei das JRuby-Projekt (*http://www.jruby.org*).

Tabelle 1.1 Java-Versionen und wichtige Änderungen im Kurzüberblick (Forts.)

Version	Jahr	Beschreibung
Java SE 8	2014	Die große Neuerung im aktuellen Java 8 sind die *Lambda-Expressions*, ein neues Sprachelement, das an vielen Stellen kürzeren und aussagekräftigeren Code ermöglicht (siehe Kapitel 11). Inoffiziell wird auch der Ausdruck *Closure* verwendet, der aber eigentlich eine etwas andere Bedeutung hat, auch dazu später mehr.
Java SE 9	2017	In Java 9 haben Sie nun die Möglichkeit, große Programme in mehrere Module aufzuteilen, wodurch Sie mehr Kontrolle darüber haben, welche Klassen für andere Teile eines Programms sichtbar sind. Auch Javas eigene Klassenbibliothek wurde in Module aufgeteilt. (mehr zum Modulsystem in Kapitel 18, »Hinter den Kulissen«)
Java 10	2019?	Über diese Version kursieren Gerüchte, dass Primitivtypen (siehe Kapitel 2, »Variablen und Datentypen«) aus der Sprache entfernt werden sollen, um Java so endlich zu 100 % objektorientiert zu machen.

Tabelle 1.1 Java-Versionen und wichtige Änderungen im Kurzüberblick (Forts.)

1.2 Die Arbeitsumgebung installieren

Bevor Sie mit Java loslegen können, müssen Sie Ihre Arbeitsumgebung einrichten. Dazu müssen Sie Java installieren und sollten Sie eine integrierte Entwicklungsumgebung (IDE) installieren, einen Editor, der Ihnen das Programmieren in Java erleichtert: Die Entwicklungsumgebung prüft schon, während Sie noch schreiben, ob der Code kompiliert werden kann, zeigt Ihnen, auf welche Felder und Methoden Sie zugreifen können, und bietet Werkzeuge, die Code umstrukturieren und sich wiederholenden Code generieren.

In diesem Buch verwenden wir die NetBeans-Entwicklungsumgebung, weil sie in einem praktischen Paket mit Java verfügbar ist. Wenn Sie eine andere IDE verwenden möchten, müssen Sie an einigen wenigen Stellen des Buches selbst die beschriebenen Funktionen finden, aber die verwendeten Funktionen stehen generell in allen Java-Entwicklungsumgebungen zur Verfügung.

Sie brauchen als Erstes das Java-Installationsprogramm. Sie finden es in den Downloads zum Buch (*www.rheinwerk-verlag.de/4096*), können aber auch unter *http://www.oracle.com/technetwork/java/javase/downloads/index.html* die aktuellste Version herunterladen.

Wenn Sie es selbst herunterladen, stellen Sie sicher, dass Sie ein JDK herunterladen, denn es gibt zwei Pakete der Java Standard Edition. JRE bezeichnet das *Java Runtime Environment*, das alles Nötige enthält, um Java-Programme auszuführen, aber nicht, um sie zu entwickeln. Dazu benötigen Sie das *Java Development Kit* (JDK), das auch die Entwicklerwerkzeuge enthält. Praktischerweise gibt es genau das auch im Paket mit NetBeans (siehe Abbildung 1.1). Achten Sie außerdem darauf, dass Sie das JDK mindestens in der Version 9 herunterladen.

Abbildung 1.1 Java-Download von der Oracle-Website

Folgen Sie dann dem Installations-Wizard für Ihr System. Wenn der Wizard fragt, ob Sie die Lizenz von JUnit akzeptieren, dann antworten Sie mit »Ja« (siehe Abbildung 1.2). Sie benötigen JUnit in Kapitel 7, »Unit Testing«.

Merken Sie sich den Pfad, unter dem Sie Java installieren; Sie benötigen ihn im nächsten Schritt.

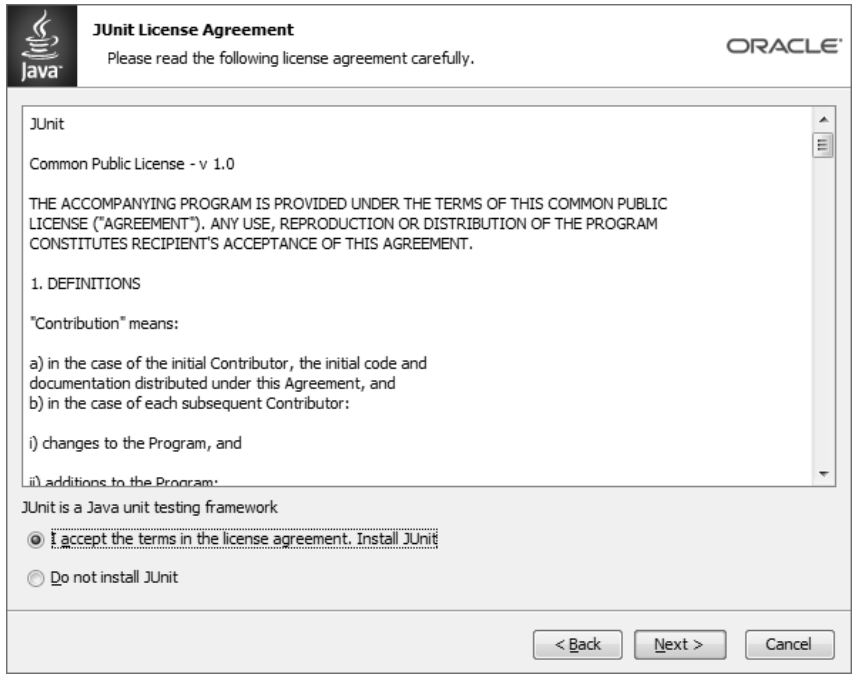

Abbildung 1.2 JUnit installieren

Sie müssen noch sicherstellen, dass die Umgebungsvariable `JAVA_HOME` gesetzt ist. Das geht leider von Betriebssystem zu Betriebssystem und von Version zu Version unterschiedlich. Finden Sie heraus, wie Sie Umgebungsvariablen für Ihr System setzen, und setzen Sie für die Variable `JAVA_HOME` den Installationspfad des JDKs als Wert. Wollen Sie Java-Programme von der Kommandozeile kompilieren und aufrufen, empfiehlt es sich außerdem, die `PATH`-Umgebungsvariable anzupassen. Dann können Sie die verschiedenen Kommandozeilentools aufrufen, ohne jedes Mal ihren vollständigen Pfad angeben zu müssen. Fügen Sie den absoluten Pfad zum Verzeichnis *bin* im JDK-Ordner hinzu.

Damit sind Sie jetzt bereit, sich in die Java-Entwicklung zu stürzen.

1.3 Erste Schritte in NetBeans

Sie haben nun das JDK und die Entwicklungsumgebung NetBeans installiert. Die wichtigsten im JDK enthaltenen Werkzeuge werden Sie im Laufe des Kapitels kennenlernen. Nun sollten Sie sich ein wenig mit NetBeans vertraut machen.

Starten Sie dazu NetBeans, und öffnen Sie das in den Downloads enthaltene Projekt WordCount, indem Sie aus dem Menü File • Open Project auswählen, in das Verzeichnis *Kapitel 1* des Downloadpakets wechseln und dort das Projekt WordCount

auswählen. Sie sollten anschließend ein zweigeteiltes Fenster sehen, ähnlich Abbildung 1.3.

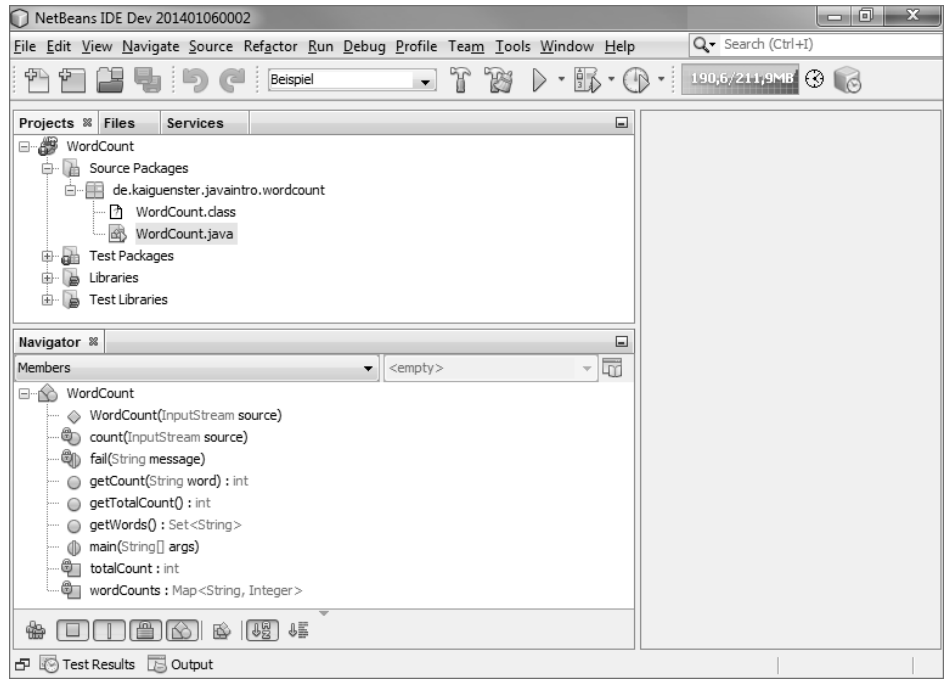

Abbildung 1.3 NetBeans: Projekt geöffnet

Der größere, rechte Bereich ist für die Codeansicht reserviert. Wenn Sie eine Java-Datei öffnen, wird ihr Inhalt hier angezeigt. Links oben sehen Sie den Tab Projects und darunter das Projekt WordCount. Falls unterhalb des Projektnamens keine weitere Anzeige zu sehen ist, öffnen Sie diese bitte jetzt durch einen Klick auf das Pluszeichen vor dem Projektnamen. Sie sollten nun unterhalb von WordCount vier Ordner sehen: Source Packages, Test Packages, Libraries und Test Libraries.

Die ersten beiden dieser Ordner enthalten den Java-Code Ihres Projekts. Unter Source Packages finden Sie den produktiven Code, das eigentliche Programm. Unter Test Packages liegen die dazugehörigen Testfälle. Die Testfälle eines Projekts sollen sicherstellen, dass der produktive Code keine Fehler enthält. Mehr zu Testfällen und testgetriebener Entwicklung erfahren Sie in Kapitel 7, »Unit Testing«. Libraries und Test Libraries enthalten Softwarebibliotheken, die Ihr produktiver Code bzw. Ihr Testcode benötigt.

Öffnen Sie den Ordner Source Packages und dort die Datei *WordCount.java*. Sie sehen jetzt in der Codeansicht den Inhalt der Datei. Außerdem sollte spätestens jetzt unterhalb des Panels Projects ein weiteres Panel mit dem Titel Navigator zu sehen sein. In diesem Panel sehen Sie Felder und Methoden der geöffneten Java-Klasse.

Durch Doppelklick auf eines dieser Elemente springen Sie in der Codeansicht sofort zu dessen Deklaration.

Die Codeansicht stellt den in der geöffneten Datei enthaltenen Quelltext dar. Zur einfachen und schnellen Übersicht werden viele Elemente farblich hervorgehoben. Das sogenannte *Syntax Highlighting* färbt Schlüsselwörter blau ein, Feldnamen grün usw. Es ist eine große Hilfe, denn viele Fehler sind sofort an der falschen Färbung erkennbar, und das Zurechtfinden im Code wird erleichtert.

Aus der Entwicklungsumgebung haben Sie außerdem die Möglichkeit, alle wichtigen Operationen auf Ihrem Code auszuführen: Durch Rechtsklick auf das Projekt in der Projektansicht können Sie es kompilieren und ein JAR-Archiv erzeugen (BUILD) sowie über Javadoc eine HTML-Dokumentation erzeugen lassen; durch Rechtsklick in die Codeansicht können Sie die Klasse ausführen oder debuggen (vorausgesetzt, sie hat eine `main`-Methode), Sie können die dazugehörigen Testfälle ausführen und vieles mehr.

Durch Rechtsklick auf einen Ordner oder ein Package in der Projektansicht haben Sie die Möglichkeit, neue Packages, Klassen, Interfaces und andere Java-Typen anzulegen. So erzeugte Java-Dateien enthalten schon die Deklaration des passenden Typs, so dass Sie diesen Code nicht von Hand schreiben müssen.

Darüber hinaus gibt es viele fortgeschrittene Funktionen, die Sie nach und nach bei der Arbeit mit der Entwicklungsumgebung kennenlernen werden.

1.4 Das erste Programm

Damit ist alles bereit für das erste Java-Programm. In den Downloads zum Buch finden Sie das NetBeans-Projekt WordCount, ein einfaches Programm, dass eine Textdatei liest und zählt, wie oft darin enthaltene Wörter jeweils vorkommen. Gleich werden wir die Elemente des Programms Schritt für Schritt durchgehen, aber zuvor sollen Sie das Programm in Action sehen. Öffnen Sie dazu das NetBeans-Projekt WordCount. In der Toolbar oben unterhalb der Menüleiste sehen Sie ein Dropdown-Menü mit ausführbaren Konfigurationen. Wählen Sie die Konfiguration BEISPIEL aus, und klicken Sie auf den grünen RUN-Button ❶ (siehe Abbildung 1.4).

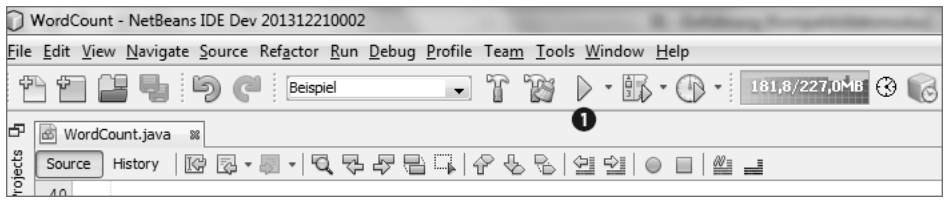

Abbildung 1.4 Das WordCount-Programm ausführen

Diese Konfiguration zählt Wortvorkommen in der mitgelieferten Datei *beispiel.txt*. Der Zählalgorithmus ist nicht perfekt, er zählt zum Beispiel »geht's« als »geht« und »s«, aber diese Sonderfälle einzubauen, würde das Programm viel komplexer machen. Die Ausgabe des Programms sieht etwa so aus wie in Abbildung 1.5.

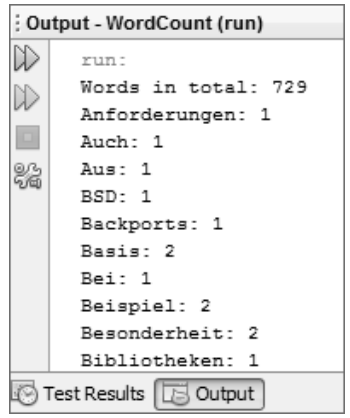

Abbildung 1.5 Die Ausgabe von WordCount

Um eine andere Datei zu verarbeiten, muss der Aufrufparameter des Programms geändert werden. Das geht entweder über den Punkt CUSTOMIZE... in der Konfigurationsauswahl oder indem Sie das Programm von der Kommandozeile aus aufrufen, dazu mehr am Ende dieses Abschnitts.

Jetzt aber zum Programmcode. Java-Programme bestehen immer aus einer oder mehreren Klassen. Genaues zu Klassen erfahren Sie, wenn wir zur Objektorientierung kommen. Für jetzt reicht es, zu wissen, dass Klassen die Bausteine von Java-Programmen sind. Normalerweise wird genau eine Klasse pro Quelldatei definiert, und die Datei trägt den Namen dieser Klasse mit der Erweiterung *.java*. Das WordCount-Programm ist ein recht einfaches Programm, das mit einer Klasse und damit auch mit einer Quelldatei auskommt: der Datei *WordCount.java*. Und so sieht sie aus, Schritt für Schritt, von oben nach unten, mit Erklärungen. Machen Sie sich aber keine Sorgen, wenn Ihnen trotz der Erklärungen nicht sofort ein Licht aufgeht, was hier im Detail passiert – alles hier Verwendete werde ich im Laufe des Buches im Detail erklären.

1.4.1 Packages und Imports

```
package de.kaiguenster.javaintro.wordcount;

import java.io.*;
import java.util.*;
```

Das `package`-Statement ordnet die hier definierten Klassen in ein Package ein und muss die erste Anweisung in einer Java-Quelldatei sein, wenn es verwendet wird. Es besteht aus dem Schlüsselwort `package` und dem Package-Namen, abgeschlossen wie jedes Java-Statement mit einem Semikolon. Packages geben der Vielzahl von Java-Klassen eine Struktur, ähnlich wie Verzeichnisse es mit Dateien im Dateisystem tun. Genau wie Dateien im Dateisystem liegt eine Klasse in exakt einem Package, und ähnlich wie Verzeichnisse im Dateisystem bilden Packages eine hierarchische Struktur. Passend dazu ist es auch zwingend notwendig, dass das Package einer Klasse dem Pfad der Quelldatei entspricht. *WordCount.java* muss im Verzeichnis *de\kaiguenster\javaintro\wordcount* unterhalb des Projektverzeichnisses liegen.

Das gezeigte `package`-Statement gibt also streng genommen nicht ein Package an, sondern vier: `de`, darin enthalten `kaiguenster`, darin wiederum `javaintro` und schließlich `wordcount`. Es ist üblich, für Packages eine umgekehrte Domain-Namen-Schreibweise zu verwenden, also zum Beispiel `de.kaiguenster`, abgeleitet von der Webadresse *kaiguenster.de*.

Defaultpackage

Das `package`-Statement kann weggelassen werden, die definierten Klassen befinden sich dann im namenlosen *Defaultpackage*. Ich rate aber davon ab: Immer ein Package anzugeben, schafft mehr Übersicht und reduziert außerdem das Risiko von Namenskonflikten.

Javas eigene Klassen weichen von dieser Konvention ab, sie liegen nicht etwa im Package `com.sun.java`, sondern einfach nur im Package `java`. Die nächsten beiden Zeilen des Programms *importieren* Klassen aus zwei verschiedenen Java-Packages: `java.io` und `java.util`. Klassen, die in dieser Datei verwendet werden und nicht in demselben Package liegen, müssen mit einem `import`-Statement bekanntgemacht werden.

Der Grund dafür ist einfach: Stünden immer alle Klassen zur Verfügung, könnte es nicht mehrere Klassen des gleichen Namens in verschiedenen Packages geben, denn es wäre nicht klar, ob zum Beispiel die Klasse `java.io.InputStream` oder die Klasse `de.beispielpackage.InputStream` gemeint ist. Durch das `import`-Statement wird eindeutig erklärt, welche Klasse verwendet werden soll. Gezeigt ist hier die umfassende Variante eines Imports: `import java.io.*` importiert *alle* Klassen aus dem Package `java.io`. Es wäre mit `import java.io.InputStream` auch möglich, nur genau die Klasse `InputStream` zu importieren. Da in WordCount aber aus beiden importierten Packages mehrere Klassen verwendet werden, ist die gezeigte Schreibweise übersichtlicher.

Ausgenommen von der Importpflicht sind Klassen aus dem Package `java.lang`. Diese sind für Java so grundlegend wichtig, dass sie immer zur Verfügung stehen.

1.4.2 Klassendefinition

```
/**
 * Ein einfacher Wortzähler, der aus einem beliebigen
 * {@see InputStream} Wörter zählt. Wörter sind alle Gruppen von
 * alphabetischen Zeichen. Das führt zum Beispiel beim
 * abgekürzten "geht's" dazu, dass es als "geht" und "s" gezählt
 * wird.
 * @author Kai
 */
public class WordCount {
```

Als Nächstes folgt die Klassendefinition. Es reicht das Schlüsselwort `class`, gefolgt vom Klassennamen. Meistens steht aber, wie auch hier, ein *Access-Modifier* (Zugriffsmodifikator) davor: Das Schlüsselwort `public` führt dazu, dass diese Klasse von überall verwendet werden kann (mehr zu Access-Modifiern in Abschnitt 5.2).

Voll qualifizierte Klassennamen

Innerhalb der Klasse selbst kann immer der einfache Klassenname (zum Beispiel `WordCount`) verwendet werden. Für die Klassendefinition muss er das sogar. Manchmal muss man aber auch den *voll qualifizierten Klassennamen* (*fully qualified class name*, *FQN*) verwenden: `de.kaiguenster.javaintro.wordcount.WordCount`. Das ist vor allem dann der Fall, wenn Sie mehrere Klassen des gleichen Namens aus unterschiedlichen Packages verwenden müssen, denn sie können dann nur so auseinandergehalten werden.

Nach dem Klassennamen folgt der *Klassenrumpf* in geschweiften Klammern. In diesem Bereich steht alles, was die Klasse kann.

Der Textblock vor der Klassendefinition, eingefasst in /** und */, ist das sogenannte *Javadoc*, das Kommentare und Dokumentation zum Programm direkt im Code enthält. Javadoc ist schon im Code praktisch, da man schnell und einfach erkennen kann, was eine Klasse oder Methode tut, noch praktischer ist aber, dass man daraus eine HTML-Dokumentation zu einem Projekt erzeugen kann.

1.4.3 Instanzvariablen

Als erstes Element innerhalb des Klassenrumpfs stehen die *Klassen- und Instanzvariablen*. In WordCount gibt es keine Klassenvariablen, es werden nur zwei Instanzvariablen definiert.

```
/*In dieser HashMap werden die Vorkommen der einzelnen Wörter gezählt.*/
private Map<String, Integer> wordCounts = new HashMap<>();
```

```
//Dies ist die Gesamtwortzahl
private int totalCount = 0;
//Aus diesem Stream wird der Text gelesen
private InputStream source;
```

Variablen sind benannte, typisierte Speicherstellen. Die Variable `wordCounts` zum Beispiel verweist auf ein Objekt des Typs `java.util.Map` (`java.util.` muss dabei aber nicht explizit angegeben werden, da `java.util.*` importiert wird; es reicht, als Typ `Map` anzugeben). `Map`s werden in Java verwendet, um Zuordnungen von Schlüsseln zu Werten zu verwalten, so dass man den Wert einfach nachschlagen kann, wenn man den Schlüssel kennt. In `wordCounts` werden als Schlüssel die im Text gefundenen Wörter verwendet, als Wert die Zahl, wie oft das Wort vorkommt. Die Variable `totalCount` enthält einen `int`-Wert, den gebräuchlichsten Java-Typ für Ganzzahlen. Darin wird gezählt, wie viele Wörter der Text insgesamt enthält.

Eine Variablendeklaration besteht aus mehreren Angaben: zunächst einem Access-Modifier, der optional ist und nur bei Klassen- und Instanzvariablen möglich, nicht bei lokalen Variablen (mehr dazu in Kapitel 2, »Variablen und Datentypen«, und Abschnitt 5.2, »Access-Modifier«). Dann folgt der Typ der Variablen. Eine Variable in Java kann niemals einen anderen Typ enthalten als den in der Deklaration angegebenen; es ist sichergestellt, dass `wordCounts` nie auf etwas anderes als eine `Map` verweisen wird. `<String, Integer>` gehört noch zur Typangabe und gibt an, welche Typen in `wordCounts` enthalten sind: Es werden Strings (Zeichenketten) als Schlüssel verwendet, um Integer (Ganzzahlen) als Werte zu finden (sogenannte Typparameter werde ich in Kapitel 10, »Arrays und Collections«, erläutern).

Schließlich wird den Variablen noch ein Initialwert zugewiesen. Das passiert mit dem Gleichheitszeichen, gefolgt vom Wert. Für `totalCount` ist das einfach die Zahl 0, bei `wordCounts` wird der `new`-Operator verwendet, um ein neues Objekt vom Typ `HashMap` zu erzeugen. Eine `HashMap` ist eine bestimmte Art von `Map`, denn nichts anderes darf `wordCounts` ja zugewiesen werden. Technisch korrekt sagt man: Die *Klasse* `HashMap` implementiert das *Interface* `Map` (siehe dazu Kapitel 6, »Objektorientierung«). Auch die Zuweisung eines Initialwertes ist optional: Sie sehen, dass `source` kein Wert zugewiesen wird, das passiert erst im *Konstruktor*.

Vor allen Variablendeklarationen wird jeweils in einem Kommentar erläutert, wozu die Variable verwendet wird. Der Unterschied zwischen den beiden Schreibweisen ist nur, dass ein Kommentar, der mit // eingeleitet wird, bis zum Zeilenende geht, und ein Kommentar, der mit /* beginnt, sich über mehrere Zeilen erstrecken kann bis zum abschließenden */. Es handelt sich in beiden Fällen um einfache Codekommentare, nicht um *Javadoc*, das würde mit /** beginnen. Codekommentare sind nur im Quellcode des Programms sichtbar, sie werden im Gegensatz zu *Javadoc* nicht in die erzeugte HTML-Dokumentation übernommen. Es gehört zum guten Programmierstil, ausführlich und korrekt zu kommentieren, denn ein Programm sollte immer leicht zu lesen sein, auch und vor allem dann, wenn es schwierig zu schreiben war.

1.4.4 Der Konstruktor

Konstruktoren sind eine besondere Art von Methoden, eine benannte Abfolge von Anweisungen, die Objekte initialisieren. Vereinfacht gesagt ist eine Klasse eine Vorlage für Objekte. Eine Klasse ist im laufenden Programm genau einmal vorhanden. Aus der Vorlage können aber beliebig viele Objekte erzeugt werden, in diesem Beispiel könnte es zum Beispiel mehrere `WordCount`s geben, die die Wortanzahl verschiedener Texte enthalten. Die Aufgabe eines Konstruktors ist es, zusammen mit dem `new`-Operator, basierend auf der Klasse, Objekte zu erzeugen.

```
public WordCount(InputStream source){
    this.source = source;
}
```

Ein Konstruktor trägt immer den Namen seiner Klasse, zum Beispiel `WordCount`, und kann in runden Klammern keinen, einen oder mehrere *Parameter* deklarieren. Das sind Werte, die vom Aufrufer des Konstruktors übergeben werden müssen und die vom Konstruktor dazu verwendet werden, das gerade erzeugte Objekt zu initialisieren. Der Konstruktor von WordCount erwartet einen `java.io.InputStream` als Parameter, einen Datenstrom, der Daten aus einer Datei enthalten kann, Daten, die über eine Netzwerkverbindung geladen werden, Daten aus Benutzereingaben oder auch aus anderen Quellen. Es wäre auch möglich gewesen, eine Datei (als Datentyp `java.io.File`) als Parameter zu übergeben, aber dadurch würde WordCount auf genau diesen Fall beschränkt, der `InputStream` dagegen kann auch aus anderen Quellen stammen.

In den geschweiften Klammern steht der Rumpf des Konstruktors, also der Code, der ausgeführt wird, wenn der Konstruktor aufgerufen wird. In diesem Fall wird im Konstruktor nur der übergebene `InputStream source` der Instanzvariablen `source` zugewiesen.

1.4.5 Die Methode »count«

Hier passiert nun endlich die Arbeit, es werden Wörter gezählt. Die Deklaration der Methode `count` sieht der Deklaration des Konstruktors ähnlich, es muss aber zusätzlich angegeben werden, welchen Datentyp die Methode an ihren Aufrufer zurückgibt. Das ist hier kein echter Datentyp, sondern die spezielle Angabe `void` für »es wird nichts zurückgegeben«.

```
public void count(){
    try(Scanner scan = new Scanner(source)){
        scan.useDelimiter("[^\\p{IsAlphabetic}]");
        while (scan.hasNext()){
            String word = scan.next().toLowerCase();
            totalCount++;
            wordCounts.put(word, wordCounts.getOrDefault(word, 0) + 1);
        }
    }
}
```

In der Methode wird die Klasse `java.util.Scanner` benutzt, eines der vielen nützlichen Werkzeuge aus dem `java.util`-Package. Ein Scanner zerlegt einen Strom von Textdaten nach einstellbaren Regeln in kleine Stücke. Die Regel wird mit dem Aufruf der Methode `useDelimiter` eingestellt: Mittels eines regulären Ausdrucks (mehr dazu in Abschnitt 8.3, »Reguläre Ausdrücke«) geben wir an, dass alle nichtalphabethischen Zeichen Trennzeichen sind; dazu gehören Zahlen, Satzzeichen, Leerzeichen usw. Die Zeichenkette »Hallo Welt! Ich lerne Java« würde zerlegt in die fünf Wörter »Hallo«, »Welt«, »Ich«, »lerne« und »Java«.

Nachdem diese Einstellung gemacht ist, werden in einer `while`-Schleife (mehr zu Schleifen in Kapitel 4, »Wiederholungen«) so lange Wörter aus dem Scanner gelesen, wie dieser neue Wörter liefern kann, also zum Beispiel bis zum Dateiende. Geprüft wird das durch den Aufruf `scan.hasNext()`. Solange dieser ergibt, dass weitere Wörter zum Lesen verfügbar sind, wird der Schleifenrumpf ausgeführt. Auch der ist wieder, wie schon Klassen- und Methodenrumpf, in geschweifte Klammern gefasst.

Solange der Scanner weitere Wörter findet, wird in der Schleife das nächste Wort ausgelesen, mit der Methode `toLowerCase` in Kleinbuchstaben umgewandelt und der Variablen `word` zugewiesen. Die Umwandlung in Kleinbuchstaben ist nötig, damit »Hallo« und »hallo« nicht als unterschiedliche Wörter gezählt werden. Anschließend wird mit `totalCount++` die Gesamtzahl Wörter im Text um eins erhöht. Die letzte Zeile in der Schleife erhöht den Zähler für das gefundene Wort in `wordCounts` und rechtfertigt eine genauere Betrachtung. Um zu verstehen, was hier passiert, ist die Zeile von innen nach außen zu lesen: Zuerst wird `wordCounts.getOrDefault(word, 0)` ausgeführt. Dieser Aufruf liest den aktuellen Zähler für das Wort aus `wordCount` oder liefert 0 zurück, falls dies das erste Auftreten des Wortes ist und `wordCount` noch keinen Wert dafür enthält. Zum Ergebnis des Aufrufs wird 1 addiert, dann wird der so berechnete Wert mit `wordCounts.put` als neuer Wert für den Zähler nach `wordCounts` zurückgeschrieben.

Es folgen noch weitere Methoden, die aber keine Programmlogik enthalten. Sie dienen nur dazu, dem Benutzer eines WordCount-Objekts den Zugriff auf die ermittelten Daten zu ermöglichen, also wie viele Wörter insgesamt gefunden wurden, welche Wörter gefunden wurden und wie oft ein bestimmtes Wort vorkam. Verwendet werden sie in der `main`-Methode, dem letzten interessanten Teil des Programms.

1.4.6 Die Methode »main«

Die `main`-Methode ist eine besondere Methode in einem Java-Programm: Sie wird aufgerufen, wenn das Programm aus der Entwicklungsumgebung oder von der Kommandozeile gestartet wird. Die `main`-Methode muss immer genau so deklariert werden: `public static void main(String[] args)`. Vieles an dieser Deklaration ist Ihnen schon von den anderen Methodendeklarationen her bekannt: `public` ist ein Access-Modifier, `void` gibt an, dass die Methode keinen Wert zurückgibt, `main` ist der Methodenname, und die Methode erwartet ein `String`-Array (`String[]`, siehe Kapitel 10, »Arrays und Collections«) als Parameter. Neu ist das Schlüsselwort `static`. Eine `static`-Methode kann direkt an der Klasse aufgerufen werden, es muss nicht erst ein Objekt erzeugt werden (siehe Kapitel 5, »Klassen und Objekte«). Schauen wir uns an, was passiert, wenn Sie das Programm WordCount aufrufen:

```
public static void main(String[] args) throws FileNotFoundException {
    if (args.length != 1){
        fail("WordCount requires exactly one file name as argument");
    }
    File f = new File(args[0]);
    if (!f.exists())
        fail("File does not exist " + f.getAbsolutePath());
    if (!f.isFile())
        fail("Not a file " + f.getAbsolutePath());
    if (!f.canRead())
        fail("File not readable " + f.getAbsolutePath());
    try(FileInputStream in = new FileInputStream(f)){
        WordCount count = new WordCount(in);
        count.count();
        System.out.println("Words in total: " + count.getTotalCount());
        count.getWords().stream().sorted().forEach((word) -> {
            System.out.println(word + ": " + count.getCount(word));
        });
    }
}
```

Zunächst wird geprüft, ob dem Programm genau ein Aufrufparameter übergeben wurde, falls nicht, wird das Programm mit der Hilfsmethode `fail` mit einer Fehlermeldung abgebrochen.

Außerdem wird geprüft, ob dieser Parameter eine existierende (`f.exists()`), gültige (`f.isFile()`) und lesbare (`f.canRead()`) Datei bezeichnet, anderenfalls werden verschiedene Fehlermeldungen ausgegeben.

Sind alle Prüfungen erfolgreich, wird ein `InputStream` auf die Datei geöffnet, um sie zu lesen. Mit diesem `InputStream` wird nun endlich ein Objekt vom Typ `WordCount` erzeugt (`new WordCount(in)`). Dieses Objekt wird durch den Aufruf der Methode `count` dazu gebracht, die Wörter in der Datei zu zählen. Schließlich wird das Ergebnis ausgegeben. `System.out` bezeichnet die Standardausgabe des Programms, `System.out.println` gibt dort eine Zeile Text aus. Zunächst wird dort mit Hilfe von `count.getTotalCount()` die Gesamtwortzahl ausgegeben, dann in alphabethischer Reihenfolge die Vorkommen einzelner Wörter. Für die Ausgabe der einzelnen Wörter wird die Stream-API mit einem Lambda-Ausdruck (der Parameter für `forEach`) verwendet (mehr zu beidem in Kapitel 11, »Lambda-Ausdrücke«).

Die Formatierung mit eingerückten Zeilen, die Sie in diesem Beispiel sehen, ist übrigens die von Oracle empfohlene Art, Java-Code zu formatieren. Zur Funktion des Programms ist es nicht notwendig, Zeilen einzurücken oder auch nur Zeilenumbrüche zu verwenden, aber die gezeigte Formatierung macht den Code lesbar und übersichtlich.

1.4.7 Ausführen von der Kommandozeile

Sie haben nun gesehen, wie das Programm WordCount Schritt für Schritt funktioniert und wie Sie es aus der Entwicklungsumgebung heraus ausführen. Aber was, wenn Ihnen einmal keine Entwicklungsumgebung zur Verfügung steht? Selbstverständlich lassen sich auch Java-Programme von der Kommandozeile aus ausführen. Vorher ist allerdings noch ein Schritt notwendig, den NetBeans (und jede andere IDE) automatisch ausführt, wenn Sie den RUN-Knopf drücken: Das Programm muss zuerst kompiliert (in Bytecode übersetzt) werden.

Den Java-Compiler ausführen

Um ein Java-Programm zu kompilieren, benutzen Sie den Java-Compiler `javac`. Öffnen Sie dazu eine Konsole (im Windows-Sprachgebrauch Eingabeaufforderung), und wechseln Sie in das Verzeichnis *Kapitel01\WordCount\src*. Führen Sie den Befehl `javac de\kaiguenster\javaintro\wordcount\WordCount.java` aus. Damit dieser Aufruf funktioniert, müssen Sie die `PATH`-Umgebungsvariable erweitert haben (siehe Abschnitt 1.2, »Die Arbeitsumgebung installieren«). Haben Sie das nicht, müssen Sie statt schlicht `javac` den kompletten Pfad zu `javac` angeben.

Ist die Kompilation erfolgreich, so ist keine Ausgabe von `javac` zu sehen, und im Verzeichnis *de\kaiguenster\javaintro\wordcount* findet sich eine neue Datei *WordCount.class*. Dies ist die kompilierte Klasse `WordCount` im Java-Bytecode.

Kompilieren größerer Projekte

`javac` von Hand aufzurufen, ist für ein Projekt mit einer oder wenigen Klassen praktikabel, es wird aber bei größeren Projekten schnell unpraktisch, da alle Quelldateien angegeben werden müssen; es gibt keine Möglichkeit, ein ganzes Verzeichnis mit Unterverzeichnissen zu kompilieren. Die Entwicklungsumgebung kann das komfortabler, aber viele große Java-Projekte setzen auch ein Build-Tool ein, um das Kompilieren und weitere Schritte, zum Beispiel JARs (Java-**Ar**chive) zu packen, zu automatisieren. Verbreitete Build-Tools sind Ant (*http://ant.apache.org*) und Maven (*http://maven.apache.org*). Diese näher zu erläutern, würde den Rahmen des Buches sprengen, aber wenn Sie größere Java-Projekte angehen wollen, führt kein Weg an einem dieser Werkzeuge vorbei.

.class-Dateien zu erzeugen ist aber nicht das Einzige, was der Java-Compiler tut. Er erkennt auch eine Vielzahl an Fehlern, von einfachen Syntaxfehlern wie vergessenen Semikola oder geschweiften Klammern bis hin zu Programmfehlern wie der Zuweisung eines Objekts an eine Variable des falschen Typs. Das ist ein großer Vorteil von kompilierten gegenüber interpretierten Sprachen: Dort würde ein solcher Fehler erst auffallen, wenn die fehlerhafte Programmzeile ausgeführt wird. Fehler zur Laufzeit sind zwar auch in Java-Programmen alles andere als selten (Kapitel 9 beschäftigt sich mit Fehlern und Fehlerbehandlung), aber viele Fehler werden schon früher vom Compiler erkannt.

Das Programm starten

Im Gegensatz zu anderen kompilierten Sprachen erzeugt der Java-Compiler keine ausführbaren Dateien. Um WordCount von der Kommandozeile aus auszuführen, müssen Sie `java` aufrufen und die auszuführende Klasse übergeben. Dazu führen Sie, immer noch im Verzeichnis *src*, folgendes Kommando aus:

```
java de.kaiguenster.javaintro.wordcount.WordCount ../beispiel.txt
```

Dieser Befehl startet eine Java-VM und führt die `main`-Methode der Klasse `de.kaiguenster.javaintro.wordcount.WordCount` aus. Beachten Sie, dass dem `java`-Kommando nicht der Pfad zur `class`-Datei übergeben wird, sondern der voll qualifizierte Klassenname. Alle Backslashes sind durch Punkte ersetzt, und die Dateiendung *.class* wird nicht angegeben. Das Kommando kann nur so aufgerufen werden, es ist nicht möglich, statt des Klassennamens einen Dateipfad zu übergeben. Der zweite Parameter, `../beispiel.txt`, ist der Aufrufparameter für das WordCount-Programm, also die Datei, deren Wörter gezählt werden sollen. Alle Parameter, die nach dem Klassennamen folgen, werden an das Programm übergeben und können in dem `String[]` gefunden werden, der an `main` übergeben wird (`args` im obigen Beispiel). Parameter, die

für `java` selbst gedacht sind, stehen zwischen `java` und dem Klassennamen. Abbildung 1.6 zeigt die Ausgabe.

Abbildung 1.6 Beispielausgabe von WordCount in der Eingabeaufforderung

1.5 In Algorithmen denken, in Java schreiben

Ein Programm mit Erklärungen nachzuvollziehen, ist nicht schwer, selbst ohne Vorkenntnisse haben Sie eine ungefähre Ahnung, wie WordCount funktioniert. Aber ein erstes eigenes Programm zu schreiben, ist eine Herausforderung. Das liegt nicht in erster Linie daran, dass die Programmiersprache noch unbekannt ist, es wäre ein Leichtes, sämtliche Sprachelemente von Java auf wenigen Seiten aufzuzählen. Die Schwierigkeit ist vielmehr, in *Algorithmen* zu denken.

Definition Algorithmus

Ein Algorithmus ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems in endlich vielen Einzelschritten.

Um ein Problem mittels eines Programms zu lösen, benötigt man immer einen Algorithmus, also eine Liste von Anweisungen, die, in der gegebenen Reihenfolge ausgeführt, aus der Eingabe die korrekte Ausgabe ableitet. Diese Handlungsvorschrift muss eindeutig sein, denn ein Computer kann Mehrdeutigkeiten nicht auflösen, und sie muss aus endlich vielen Einzelschritten bestehen, denn sonst würde das Programm endlos laufen.

Die Methode `count` aus dem WordCount-Beispiel enthält einen Algorithmus, der Wortvorkommen in einem Text zählt. »Einen« Algorithmus, nicht »den« Algorithmus, denn zu jedem Problem gibt es mehrere Algorithmen, die es lösen. Einen Algorithmus zu finden, ist der schwierigere Teil des Programmierens. Wie gelangt man aber zu einem Algorithmus? Leider gibt es darauf keine universell gültige Antwort, es bedarf ein wenig Erfahrung. Aber die folgenden zwei Beispiele geben Ihnen einen ersten Einblick, wie ein Algorithmus entwickelt und anschließend in Java umgesetzt wird.

1.5.1 Beispiel 1: Fibonacci-Zahlen

Die Fibonacci-Folge ist eine der bekanntesten mathematischen Folgen überhaupt. Elemente der Folge werden berechnet, indem die beiden vorhergehenden Elemente addiert werden: $fibonacci_n = fibonacci_{n-1} + fibonacci_{n-2}$. Die ersten zehn Zahlen der Folge lauten 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

Da mathematische Formeln eine Schreibweise für Algorithmen sind – sie erfüllen die oben angegebene Definition –, ist es sehr einfach, daraus eine Liste von Operationen zu erstellen, die anschließend in ein Java-Programm umgesetzt werden können.

Berechnung der n-ten Fibonacci-Zahl

1. Berechne die (n–1)-te Fibonacci-Zahl.
2. Berechne die (n–2)-te Fibonacci-Zahl.
3. Addiere die Ergebnisse aus 1. und 2., um die n-te Fibonacci-Zahl zu erhalten.

In Schritt 1 und 2 ruft der Algorithmus sich selbst auf, um die (n–1)-te und (n–2)-te Fibonacci-Zahl zu berechnen. Das widerspricht nicht der Definition, denn wenn der Algorithmus keine Fehler enthält, schließt die Berechnung trotzdem in endlich vielen Schritten ab. Einen Algorithmus, der sich in dieser Art auf sich selbst bezieht, nennt man *rekursiv*.

Der Algorithmus wie gezeigt ist aber noch nicht vollständig. Die 0. und 1. Fibonacci-Zahl können nicht nach dieser Rechenvorschrift berechnet werden, ihre Werte sind als 0 und 1 festgelegt. Wenn Sie selbst Fibonacci-Zahlen berechnen und dies nicht Ihr erster Kontakt mit der Fibonacci-Folge ist, dann haben Sie diese Information als Vorwissen. Ein Algorithmus hat aber kein Vorwissen. Dies ist eine wichtige Grundlage bei der Entwicklung von Algorithmen: Ein Algorithmus hat niemals Vorwissen. Der gezeigte Algorithmus würde versuchen, die 0. Fibonacci-Zahl zu berechnen, indem er die –1. und die –2. Fibonacci-Zahl addiert usw. Der Algorithmus würde nicht enden, sondern würde sich immer weiter in die negativen Zahlen begeben. Damit der Algorithmus vollständig und korrekt ist, muss er also die beiden Startwerte der Folge berücksichtigen. Außerdem ist das Ergebnis für $n < 0$ nicht definiert, in diesem Fall sollte die Berechnung also sofort mit einer Fehlermeldung abgebrochen werden. So ergibt sich der korrekte Algorithmus.

Berechnung der n-ten Fibonacci-Zahl

1. Wenn n < 0, dann gib eine Fehlermeldung aus.
2. Wenn n = 0, dann ist die n-te Fibonacci-Zahl 0.
3. Wenn n = 1, dann ist die n-te Fibonacci-Zahl 1.
4. Wenn n > 1, dann:
   - Berechne die (n–1)-te Fibonacci-Zahl.
   - Berechne die (n–2)-te Fibonacci-Zahl.
   - Addiere die beiden Zahlen, um die n-te Fibonacci-Zahl zu erhalten.

Dieser Algorithmus kann nun korrekt alle Fibonacci-Zahlen berechnen. Die Umsetzung des Algorithmus in Java ist nun vergleichsweise einfach, es wird Schritt für Schritt der obige Pseudocode umgesetzt. So entsteht die folgende Java-Methode zur Berechnung von Fibonacci-Zahlen:

```
public static int fibonacci(int n){
    if (n < 0)
         throw new IllegalArgumentException("Fibonacci-Zahlen sind für
           negativen Index nicht definiert.");
    else if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Listing 1.1 Berechnung von Fibonacci-Zahlen

Es wird eine Methode mit dem Namen `fibonacci` definiert, die eine Ganzzahl als Parameter erwartet. Die Methodendeklaration kennen Sie schon aus dem WordCount-Beispiel. Innerhalb der Methode werden dieselben vier Fälle geprüft wie im Pseudocode. In Java wird das `if`-Statement benutzt, um Wenn-dann-Entscheidungen zu treffen. Die `else`-Klausel des Statements gibt einen Sonst-Fall an. Die Bedingung, nach der `if` entscheidet, wird in Klammern geschrieben. Die Vergleiche »größer als« und »kleiner als« funktionieren genau, wie man es erwartet, die Prüfung auf Gleichheit muss mit einem doppelten Gleichheitszeichen erfolgen. Der Code wird von oben nach unten ausgeführt. Zuerst wird geprüft, ob n < 0 ist, falls ja, wird mit `throws` ein Fehler, in Java *Exception* genannt, »geworfen« (mehr dazu in Kapitel 9, »Fehler und Ausnahmen«). Anderenfalls wird als Nächstes geprüft, ob n = 0 ist. In diesem Fall wird mit `return` das Ergebnis 0 zurückgegeben, das heißt, der Aufrufer der Methode erhält als Resultat seines Aufrufs den Wert 0. Traf auch das nicht zu, wird als Nächstes n = 1

geprüft und im Positivfall 1 zurückgegeben. Schließlich, wenn alle anderen Fälle nicht zutreffen, wird auf die Rechenvorschrift für Fibonacci-Zahlen zurückgegriffen: Die `fibonacci`-Methode wird mit den Werten n–1 und n–2 aufgerufen, die Ergebnisse werden addiert, und die Summe wird als Gesamtergebnis zurückgegeben.

Damit ist die Berechnung vollständig, aber eine Methode kann in Java nicht für sich stehen, sie **muss** immer zu einer Klasse gehören. Der Vollständigkeit halber sehen Sie hier die Klasse `Fibonacci`:

```
public class Fibonacci {

    public static int fibonacci(int n){…}

    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println(
            "Aufruf: java de.kaiguenster.javaintro.fibonacci.Fibonacci <n>");
            System.exit(1);
        }
        int n = Integer.parseInt(args[0]);
        int result = fibonacci(n);
        System.out.println("Die " + n + ". Fibonacci-Zahl ist: " + result);
    }
}
```

Die Klassendeklaration ist Ihnen bereits bekannt. In der `main`-Methode wird wieder zuerst geprüft, ob die richtige Zahl von Aufrufparametern übergeben wurde, anschließend wird mit diesen Parametern die `fibonacci`-Methode aufgerufen und schließlich das Ergebnis ausgegeben.

Neu ist lediglich die Umwandlung einer Zeichenkette in eine Zahl. Die Aufrufparameter liegen immer als Zeichenkette vor, die Methode `fibonacci` benötigt aber eine Zahl als Eingabe. Diese Umwandlung leistet die Zeile `int n = Integer.parseInt(args[0])`: Der erste Aufrufparameter wird an die Methode `Integer.parseInt` übergeben, die genau diese Umwandlung durchführt. Das Ergebnis, nun eine Zahl, wird der Variablen `n` zugewiesen und dann endlich an die Methode `fibonacci` übergeben.

1.5.2 Beispiel 2: Eine Zeichenkette umkehren

Mathematische Formeln sind wie gesehen nicht schwer in ein Java-Programm umzusetzen. Aber was, wenn eine andere Art von Problem zu lösen ist? Als zweites Beispiel wollen wir eine beliebige Zeichenkette Zeichen für Zeichen umdrehen. Aus »Hallo Welt!« soll zum Beispiel »!tleW ollaH« werden.

Ein Algorithmus für diese Aufgabe ist sofort offensichtlich: Angefangen bei einem leeren Ergebnis wird das letzte Zeichen der Eingabe abgeschnitten und so lange an das Ergebnis gehängt, bis die Eingabe abgearbeitet ist. Leider ist das Abschneiden des letzten Zeichens in Java nicht die einfachste Lösung, dadurch sieht der Algorithmus ein wenig umständlicher aus.

Eine Zeichenkette umkehren

1. Beginne mit einer leeren Zeichenkette als Ergebnis.
2. Lies die Eingabe von hinten nach vorn. Für jedes Zeichen der Eingabe:
   - Hänge das Zeichen hinten an das Ergebnis an.
3. Gib das Ergebnis aus.

Der Algorithmus ist sogar einfacher als die Berechnung der Fibonacci-Zahlen. Das spiegelt sich auch im Java-Code wider:

```
public static String reverse(String in){
    if (in == null)
        throw new IllegalArgumentException("Parameter in muss
        übergeben werden.");
    StringBuilder out = new StringBuilder();
    for (int i = in.length() - 1; i >= 0; i--){
        out.append(in.charAt(i));
    }
    return out.toString();
}
```

Listing 1.2 Eine Zeichenkette umkehren

Die Implementierung des Algorithmus ist kurz und bündig. Die Methode `reverse` erwartet als Eingabe eine Zeichenkette, in Java *String* genannt. Auch in dieser Methode prüft die erste Anweisung, ob der übergebene Parameter einen gültigen Wert hat. Die sogenannte defensive Programmierung, also die Angewohnheit, Eingaben und Parameter immer auf ihre Gültigkeit hin zu überprüfen, ist guter Stil für eine fehlerfreie Software. Hier wird geprüft, ob der übergebene String den speziellen Wert `null` hat. `null` bedeutet in Java, dass eine Variable keinen Wert hat, sie ist nicht mit einem Objekt gefüllt. Es ist wichtig, dass Sie den Wert `null` und die Zeichenkette `"null"`, in Anführungszeichen, auseinanderhalten. Der String `"null"` ließe sich natürlich zu `"llun"` umkehren, aber `null` ist kein String und somit keine gültige Eingabe für die Methode.

Als Nächstes wird die Ergebnisvariable vom Typ `StringBuilder` initialisiert. Wie der Name schon klarmacht, ist die Klasse `StringBuilder` dazu da, Strings zu bauen. Es wäre auch möglich, als Ergebnis direkt einen String zu verwenden, aber wie Sie später

sehen werden, hat der `StringBuilder` Vorteile, wenn ein String wie hier aus Fragmenten oder einzelnen Zeichen zusammengesetzt wird.

Es folgt die eigentliche Arbeit der Methode: den String umzukehren. Dazu kommt eines der in Java zur Verfügung stehenden Schleifenkonstrukte zum Einsatz: die `for`-Schleife. Allen Schleifen ist gemein, dass ihr Rumpf mehrfach ausgeführt wird. Die Besonderheit der `for`-Schleife ist, dass sie eine Zählvariable zur Verfügung stellt, die zählt, wie oft die Schleife bereits durchlaufen wurde. Diese Variable wird traditionell schlicht `i` benannt und zählt in diesem Fall von der Anzahl Zeichen im Eingabe-String bis 0 herunter. In einem String mit fünf Zeichen hat `i` im ersten Schleifendurchlauf den Wert 4, danach die Werte 3, 2, 1 und schließlich den Wert 0. Der Wert im ersten Durchlauf ist 4, nicht 5, weil das erste Zeichen im String den Index 0 hat, nicht 1. In jedem Durchlauf wird mit `in.charAt(i)` das `i`-te Zeichen aus der Eingabe ausgelesen und mit `out.append()` an das Ergebnis angehängt.

Nach dem Ende der Schleife wird aus `out` durch die Methode `toString()` ein String erzeugt und dieser mit `return` an den Aufrufer zurückgegeben.

Auch in diesem Beispiel ist die Methode, die die Aufgabe löst, nicht alles. Dazu gehören wieder eine Klassendeklaration und eine `main`-Methode, die aber keine großen Neuerungen mehr enthalten:

```
public static void main(String[] args) {
    if (args.length != 1){
        System.out.println("Aufruf: java de.kaiguenster.javaintro
                                          .reverse.Reverse <text>");
        System.exit(1);
    }
    String reversed = reverse(args[0]);
    System.out.println(reversed);
}
```

Listing 1.3 Die »main«-Methode

Für den Aufruf des Programms ist nur noch ein Sonderfall zu beachten: Wenn im Eingabe-String Leerzeichen vorkommen, dann müssen Sie ihn beim Aufruf in Anführungszeichen setzen. Ohne Anführungszeichen wäre jedes Wort ein eigener Eintrag in `args`; mit den Anführungszeichen wird die gesamte Zeichenkette als ein Parameter behandelt und so zu einem Eintrag in `args`.

1.5.3 Algorithmisches Denken und Java

Diese zwei Beispiele zeigen, dass die Umsetzung eines Algorithmus in Java keine Schwierigkeit ist, wenn Sie die passenden Sprachelemente kennen. In den nächsten Kapiteln werden Sie sämtliche Sprachelemente von Java kennenlernen.

Anspruchsvoller ist es, einen Algorithmus zur Lösung eines Problems zu entwerfen. Dieser Teil des Programmierens kann nur begrenzt aus einem Buch gelernt werden, es gehört Erfahrung dazu. Die Möglichkeiten der Sprache zu kennen, hilft allerdings sehr dabei, einen effektiven Algorithmus zu entwerfen, und durch die Beispiele und Aufgaben in diesem Buch werden Sie Muster kennenlernen, die Sie immer wieder einsetzen können, um neue Probleme zu lösen.

1.6 Die Java-Klassenbibliothek

In den vorangegangenen Beispielen haben Sie schon einige Klassen aus der Java-Klassenbibliothek kennengelernt. Dieser kleine Einblick kratzt jedoch kaum an der Oberfläche, die Klassenbibliothek von Java 9 enthält mehr als 4.000 Klassen. Glücklicherweise ist die Klassenbibliothek in Packages unterteilt. Wenn Sie die wichtigsten Packages und ihre Aufgaben kennen, so haben Sie dadurch einen guten Ausgangspunkt, nützliche Klassen zu finden. Tabelle 1.2 listet Ihnen die wichtigsten Packages auf.

Package	Inhalt
`java.lang`	Das Package `java.lang` enthält die Grundlagen der Plattform. Hier finden Sie zum Beispiel die Klasse `Object`, von der alle anderen Klassen erben (siehe Kapitel 5, »Klassen und Objekte«), aber auch die grundlegenden Datentypen wie `String` und die verschiedenen Arten von `Number` (`Integer`, `Float` ...). Ebenfalls hier zu finden sind die Klasse `Thread`, die parallele Programmierung in Java ermöglicht (siehe Kapitel 13, »Multithreading«), und einige Klassen, die direkt mit dem Betriebssystem interagieren, allen voran die Klasse `System`. Außerdem finden Sie hier die Klasse `Math`, die Methoden für mathematische Operationen über die vier Grundrechenarten hinaus enthält (Potenzen, Wurzeln, Logarithmen, Winkeloperationen usw.). Eine Besonderheit des `java.lang`-Packages ist es, dass die darin enthaltenen Klassen nicht importiert werden müssen, `java.lang.*` steht immer ohne Import zur Verfügung.
`java.util`	Dieses Package und seine Unter-Packages sind eine Sammlung von diversen Werkzeugen, die in vielen Programmen nützlich sind. Dazu gehören zum Beispiel Listen und Mengen, ein Zufallszahlengenerator, `Scanner` und `StringTokenizer` zum Zerlegen einer Zeichenkette und vieles mehr.

Tabelle 1.2 Die wichtigsten Packages der Java 8 Standard Edition

Package	Inhalt
`java.util` (Forts.)	In Unter-Packages finden Sie unter anderem reguläre Ausdrücke (`java.util.regex`), Werkzeuge zum Umgang mit ZIP-Dateien (`java.util.zip`) und eine Werkzeugsammlung zum Schreiben konfigurierbarer Logdateien (`java.util.logging`). Viele Klassen aus diesem Package werden Sie im Laufe des Buches kennenlernen, vor allem in Kapitel 8, »Die Standardbibliothek«.
`java.awt`	`java.awt` und seine Unter-Packages enthalten das erste und älteste von inzwischen drei Frameworks, mit denen Sie grafische Benutzeroberflächen in Java entwickeln können. Die neueren Alternativen sind das *Swing-Framework* aus dem Package `javax.swing` und das neue *JavaFX*, das sich bereits großer Beliebtheit erfreut. Mehr zu JavaFX erfahren Sie in Kapitel 16, »GUIs mit JavaFX«.
`java.io`	Unter `java.io` finden Sie Klassen, die Lesen aus und Schreiben in Dateien ermöglichen. Dabei sind die verschiedenen Varianten von `InputStream` und `OutputStream` für das Lesen bzw. Schreiben von Binärdateien zuständig, `Reader` und `Writer` erledigen dieselben Aufgaben für Textdateien. Es gibt auch Varianten von allen vier Klassen, die Daten aus anderen Quellen lesen bzw. dorthin schreiben, diese finden Sie aber in anderen Packages. Details zu I/O (Input/Output) in Java finden Sie in Kapitel 12, »Dateien, Streams und Reader«.
`java.nio`	NIO steht für *New I/O* oder *Non-Blocking I/O*, je nachdem, welcher Quelle man glaubt. Es handelt sich hier um eine Alternative zum `java.io`-Package. Traditionelles I/O wird auch als *Blocking I/O* bezeichnet, weil ein Thread auf das Ergebnis dieser Operationen warten muss. Bei Anwendungen mit sehr vielen und/oder großen I/O-Operationen führt dies zu Problemen. Mit NIO werden keine Threads mit Warten blockiert, die Anwendung ist stabiler und performanter. Allerdings ist das Programmiermodell von NIO um vieles komplexer als traditionelles I/O, und nur die wenigsten Anwendungen bedürfen wirklich der gebotenen Vorteile.
`java.math`	Das kleinste Package in der Klassenbibliothek, `java.math`, enthält nur vier Klassen. Die zwei wichtigen Klassen, `BigInteger` und `BigDecimal`, dienen dazu, mit beliebig großen und beliebig genauen Zahlen zu arbeiten.

Tabelle 1.2 Die wichtigsten Packages der Java 8 Standard Edition (Forts.)

Package	Inhalt
`java.math` (Forts.)	Die regulären numerischen Typen wie `int` und `float` unterliegen Begrenzungen, was Größe und Genauigkeit angeht (dazu mehr in Kapitel 2, »Variablen und Datentypen«), `BigInteger` und `BigDecimal` sind nur vom verfügbaren Speicher begrenzt.
`java.net`	Enthält Klassen zur Netzwerkkommunikation. Hier finden Sie Klassen, mit denen Sie Kommunikation per UDP oder TCP als Client oder als Server realisieren. Ebenfalls hier enthalten sind Klassen, um auf einer höheren Abstraktionsebene mit URLs und Netzwerkprotokollen zu arbeiten.
`java.security`	Unter `java.security` finden Sie Werkzeuge für zwei sehr unterschiedliche Sicherheitsbelange. Einerseits sind das Klassen für kryptografische Operationen (Verschlüsselung, Signierung, Key-Generierung), andererseits ein sehr mächtiges, konfigurierbares *Permission-Framework*, mit dem sich sehr fein abstimmen lässt, auf welche Ressourcen eine Java-Anwendung welche Art von Zugriff haben soll.
`java.sql`	Dieses Package dient der Anbindung von Java-Anwendungen an SQL-Datenbanken wie MySQL. Sie benötigen dazu einen Datenbanktreiber, der inzwischen von allen Datenbankherstellern zur Verfügung gestellt wird, und können dann sehr einfach SQL-Anfragen an die Datenbank stellen.
`java.text`	Hilfsmittel zur Textformatierung. Mit den hier enthaltenen Klassen können Sie Zahlen und Daten konfigurierbar und sprachabhängig formatieren, zum Beispiel um in einer mehrsprachigen Anwendung das zur Sprache passende Datumsformat zu verwenden. Ebenso können Sie im regionalen Format eingegebene Daten in die entsprechenden Java-Datentypen umwandeln.
`java.time`	Dieses in Java 8 neu hinzugekommene Package bietet Datentypen und Werkzeuge für die Arbeit mit Zeit- und Datumswerten sowie für den Umgang mit Zeitzonen.
`javax.swing`	Das zweite Package für das Erstellen grafischer Benutzeroberflächen. Das Swing-Framework ist moderner und leichtgewichtiger als das ältere AWT. Inzwischen ist aber auch Swing nicht mehr das neueste GUI-Framework, dieser Titel gebührt nun JavaFX.
`javafx`	In diesem Package liegt das neueste GUI-Framework von Java, das ich in Kapitel 16, »GUIs mit JavaFX«, ausführlich besprechen werde.

Tabelle 1.2 Die wichtigsten Packages der Java 8 Standard Edition (Forts.)

Dies sind nur die wichtigsten Packages der Klassenbibliothek, es gibt noch viele weitere, vor allem im Haupt-Package `javax`, dessen Einsatzgebiet aber spezieller ist. Auch haben viele der aufgeführten Packages wiederum Unter-Packages. Eine komplette Übersicht über alle verfügbaren Packages und Klassen bietet Ihnen die aktuelle Version des *Javadocs*.

1.7 Dokumentieren als Gewohnheit – Javadoc

Sie haben Javadoc oben bereits kennengelernt – es dient dazu, Ihre Programme direkt im Quelltext zu dokumentieren und daraus ganz einfach eine HTML-Dokumentation zu erzeugen. Dieses Instrument zur Dokumentation wird auch wirklich genutzt, die meisten Open-Source-Bibliotheken in Java enthalten auch Javadoc-Dokumentation. Und die Java-Klassenbibliothek geht mit gutem Beispiel voran: Jede Klasse und Methode ist ausführlich beschrieben. Wenn Sie also Fragen haben, wie eine Klasse der Standardbibliothek zu verwenden ist, oder sich einfach umschauen wollen, welche Möglichkeiten Ihnen Java noch bietet, dann ist das Javadoc der richtige Ort. Sie finden es für die aktuelle Version 9 unter der URL *http://download.java.net/jdk9/docs/api/*.

1.7.1 Den eigenen Code dokumentieren

Wie Sie in den Beispielen schon gesehen haben, ist es sehr einfach, mit Javadoc zu dokumentieren. Es reicht, die Dokumentation zwischen /** und */ zu fassen. Aber Javadoc bietet Möglichkeiten über diese einfache Dokumentation hinaus. Sie können in Ihrer Dokumentation HTML-Tags benutzen, um den Text zu formatieren, zum Beispiel können Sie mit dem `<table>`-Tag eine Tabelle integrieren, wenn Sie tabellarische Daten in der Dokumentation aufführen. Außerdem gibt es eine Reihe spezieller Javadoc-Tags, die bestimmte Informationen strukturiert wiedergeben. Diese Tags sind mit dem Präfix @ markiert. Betrachten Sie zum Beispiel die Dokumentation der `reverse`-Methode aus dem obigen Beispiel:

```
/**
 * Kehrt einen <code>String</code> zeichenweise um. Zum Beispiel
 * wird "Hallo, Welt!" zu "!tleW ,ollaH"
 * @param in - der umzukehrende <code>String</code>
 * @return den umgekehrten <code>String</code>
 * @throws IllegalArgumentException wenn in == <code>null</code>
 */
```

Listing 1.4 Javadoc einer Methode

Und so wie in Abbildung 1.7 sieht die daraus erzeugte Dokumentation im HTML-Format aus.

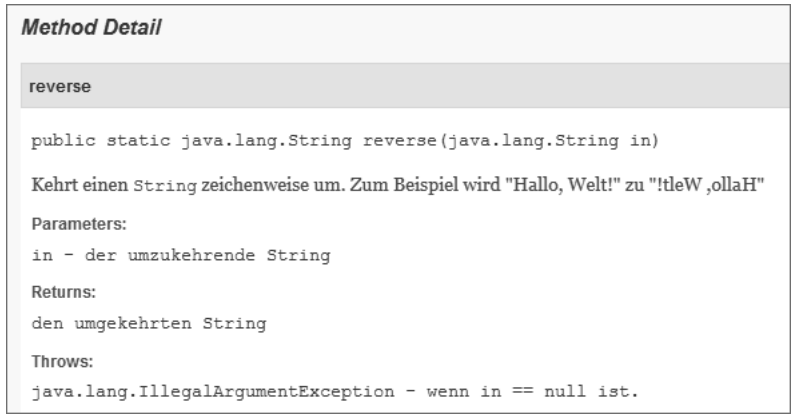

Abbildung 1.7 Die generierte HTML-Dokumentation

Hier wurden drei Javadoc-Tags verwendet: `@param`, `@return` und `@throws`. Alle drei dokumentieren Details über die Methode (siehe Tabelle 1.3).

Tag	Bedeutung
`@param`	Beschreibt einen Parameter der Methode. Dem Tag `@param` muss immer der Name des Parameters folgen, anschließend dessen Bedeutung. Bei Methoden mit mehreren Parametern kommt `@param` für jeden Parameter einmal vor.
`@return`	Beschreibt den Rückgabewert der Methode. Dieses Tag darf nur einmal auftreten.
`@throws` (oder `@exception`)	Beschreibt, welche Fehler die Methode werfen kann und unter welchen Umständen. Es ist nicht nötig, alle Fehler zu dokumentieren, die möglicherweise auftreten könnten. Üblicherweise führen Sie nur solche Fehler auf, die Sie selbst mit `throw` werfen, und solche, die als *Checked Exception* (siehe Kapitel 9, »Fehler und Ausnahmen«) deklariert wurden. Auch dieses Tag kann mehrfach auftreten.

Tabelle 1.3 Javadoc-Tags speziell für Methoden

Die Bedeutung von Parametern und Rückgabewert ist häufig schon aus dem Beschreibungstext ersichtlich, deswegen ist es hier in Ordnung, keinen vollständigen Satz anzugeben oder die Beschreibung sogar gänzlich leer zu lassen. Das `@param`-Tag muss aber für jeden Parameter vorhanden sein.

Für diese Elemente Tags zu verwenden, anstatt sie nur im Text zu beschreiben, hat den Vorteil, dass manche Entwicklerwerkzeuge die Beschreibungen auswerten und zum Beispiel den Text des `@param`-Tags anzeigen, wenn Sie gerade diesen Parameter eingeben.

Diese Tags ergeben natürlich nur im Kontext von Methoden einen Sinn, denn nur diese haben Parameter und Rückgabewerte, und nur diese werfen Fehler. Es gibt einige andere Tags, die nur bei Klassen (und äquivalenten Elementen wie Interfaces, Enums etc.) zum Einsatz kommen.

```
/**
 * Programm zum Umkehren von Strings in der Kommandozeile.
 * @author Kai
 * @version 1.0
 */
```

Listing 1.5 Javadoc einer Klasse

Die Tags in Tabelle 1.4 sind eher als Metainformationen zu verstehen. Sie haben keine direkte Relevanz für das Arbeiten mit der Klasse.

Tag	Bedeutung
`@author`	Nennt den oder die Autoren der Klasse. Für mehrere Autoren wird das `@author`-Tag wiederholt.
`@version`	Die aktuelle Version des vorliegenden Quellcodes. Die Version muss keinem bestimmten Format folgen, sie kann eine typische Versionsnummer (zum Beispiel »1.3.57«) oder eine fortlaufende Nummer sein, aber auch ein beliebiger anderer Text.

Tabelle 1.4 Beispiele für Javadoc-Tags mit Metainformationen

Über diese speziellen Tags für Methoden und Klassen hinaus gibt es allgemeine Tags, die an beliebigen Elementen erlaubt sind (siehe Tabelle 1.5).

Tag	Bedeutung
`@deprecated`	Dieses wichtige Tag markiert ein Element, das aktuell noch existiert, aber in einer zukünftigen Version entfernt werden wird. Der Text hinter dem `@deprecated`-Tag kann erläutern, warum das Element entfernt wird, und vor allem, was stattdessen genutzt werden kann.

Tabelle 1.5 Allgemeine Javadoc-Tags

Tag	Bedeutung
`@deprecated` (Forts.)	*Deprecation* stellt einen Weg zur sanften Migration von APIs dar: Anwendungen, die mit einer alten Version einer API entwickelt wurden, funktionieren mit der neuen Version weiter, aber dem Entwickler wird ein Hinweis gegeben, dass er die Anwendung anpassen sollte, weil eine zukünftige Version der API nicht mehr kompatibel sein wird. Seit Java 5 kann auch die `@Deprecated`-Annotation genutzt werden (Näheres zu Annotationen in Kapitel 6, »Objektorientierung«), aber das Javadoc-Tag wird nach wie vor unterstützt.
`@see`	Erzeugt einen »Siehe auch«-Link am Ende der Dokumentation. Als Wert des `@see`-Tags kann Text in Anführungszeichen angegeben werden, ein HTML-Link (`<a href="...">`) oder eine Klasse, Methode oder ein Feld aus dem aktuellen Projekt. Das Format für den letzten Fall wird im Kasten unter dieser Tabelle beschrieben.
`@link`	Dieses Tag erfüllt eine ähnliche Funktion wie das `@see`-Tag, kann aber im Fließtext verwendet werden. Das Format für das Linkziel entspricht dem von `@see`. Um das Linkziel vom Fließtext abzugrenzen, müssen das `@link`-Tag und sein Wert in geschweifte Klammern gefasst werden, zum Beispiel so: `{@link java.lang.Object# toString}`.
`@since`	`@since` gibt an, seit welcher Version ein Element verfügbar ist. Dieses Tag ist besonders wertvoll im Javadoc des JDKs selbst, denn es kennzeichnet, seit welcher Java-Version eine Klasse oder Methode existiert.

Tabelle 1.5 Allgemeine Javadoc-Tags (Forts.)

Javadoc: Links auf andere Elemente

Links mit `@see` und `@link` auf ein anderes Programmelement müssen einem bestimmten Format folgen. Voll ausgeschrieben lautet dieses Format: *<voll qualifizierter Klassenname>#<Feld oder Methode>*, zum Beispiel: `@see de.kaiguenster.javaintro.reverse.Reverse#reverse(String)`, was einen Link auf die Methode `reverse` der Klasse `Reverse` erzeugt. Genau wie im Java-Code kann der einfache Klassenname ohne Package verwendet werden, wenn die Zielklasse importiert wird: `@see Reverse#reverse(String)`. Für Links auf eine Klasse statt auf ein Feld oder eine Methode der Klasse entfallen das #-Zeichen und der darauffolgende Text. Bei Links auf eine Methode oder ein Feld in derselben Klasse kann die Klassenangabe entfallen: `@see #reverse(String)`. Für Links auf Methoden müssen wie gezeigt die Parametertypen der Methode aufgelistet werden. Das ist deshalb notwendig, weil eine Java-Klasse mehrere Methoden mit demselben Namen enthalten kann, die sich nur durch ihre Parametertypen unterscheiden. Für einen Link auf ein Feld (eine Klassen- oder Instanzvariable) wird hinter der Raute nur der Name angegeben.

1.7.2 Package-Dokumentation

Genau wie Klassen, Methoden und Felder lassen sich auch Packages mit Javadoc dokumentieren. Da Packages aber nicht in einer eigenen Datei liegen, muss für die Dokumentation eine Datei angelegt werden. Dazu gibt es zwei Möglichkeiten.

Für reine Dokumentationszwecke kann eine HTML-Datei mit dem Namen *package.html* im zum Package passenden Verzeichnis abgelegt werden. Im `<body>`-Element dieser HTML-Datei steht die Beschreibung des Packages, es stehen dort alle Möglichkeiten zur Verfügung, die auch an anderen Stellen im Javadoc erlaubt sind.

Es gibt auch die neuere Möglichkeit, eine vollwertige Package-Deklaration in einer eigenen Quellcodedatei namens *package-info.java* zu verwenden. In dieser Datei steht nur ein `package`-Statement mit einem Javadoc-Block, wie er auch an einer Klasse angegeben würde. Der Vorteil dieser Variante ist, dass auch *Annotationen* für das Package angegeben werden können – ein Vorteil, der, zugegeben, nur selten zum Tragen kommt.

```
/**
 * Hier steht die Package-Beschreibung.
 */
package de.kaiguenster.beispiel;
```

Listing 1.6 Eine Package-Deklaration: »package-info.java«

1.7.3 HTML-Dokumentation erzeugen

Wenn die Dokumentation im Quellcode vorliegt, lässt sich darauf schnell und einfach die HTML-Dokumentation erzeugen. Das dafür benötigte Programm `javadoc` wird mit dem JDK ausgeliefert. Um die Dokumentation eines Projekts zu erzeugen, sieht der Aufruf typischerweise so aus:

```
javadoc -d C:\javadoc -charset UTF-8 -subpackages de
```

Der Parameter `-d` gibt dabei an, wo die generierten HTML-Dateien abgelegt werden. `-charset` legt den Zeichensatz fest, in dem die Quelldateien gespeichert sind. Vor al-

lem unter Windows ist dieser Parameter wichtig, denn für NetBeans und die meisten anderen Entwicklungsumgebungen ist UTF-8 der Standardzeichensatz, Windows verwendet aber einen anderen Zeichensatz als Default. Ohne `-charset` werden Umlaute nicht korrekt in die HTML-Dateien übernommen. Der Parameter `-subpackages` bedeutet schließlich, dass Dokumentation für das Package `de` und alle darunterliegenden Packages erzeugt werden soll.

Das `javadoc`-Programm bietet eine Vielzahl weiterer Optionen, mit denen Sie steuern, aus welchen Quelldateien Dokumentation erzeugt werden soll und wie die Ausgabe aussieht. Diese Optionen können Sie in der Dokumentation des Programms nachlesen. An dieser Stelle erwähnenswert sind lediglich noch die vier Optionen `-public`, `-protected`, `-package` und `-private`. Sie entsprechen den vier Access-Modifiern (`public`, `protected`, `private` und ohne Angabe für Package-Sichtbarkeit, siehe Abschnitt 5.2, »Access-Modifier«). Wird einer dieser Parameter angegeben, so wird Dokumentation für alle Klassen, Methoden und Felder generiert, die diesen Access-Modifier oder einen »öffentlicheren« haben. Der Default-Wert, falls keiner dieser Parameter gesetzt wird, ist `-protected`. Das bedeutet, dass Dokumentation für alle Programmelemente die Sichtbarkeit `public` oder `protected` haben. Eine komplette Dokumentation aller Elemente entsteht mit `-private`.

1.7.4 Was sollte dokumentiert sein?

Sie haben nun gesehen, wie Sie Ihren Java-Code dokumentieren können. Aber **was** sollte dokumentiert werden? Welche Programmelemente sollten Sie mit Javadoc versehen?

Wie bei so vielen Dingen gibt es auch darüber unterschiedliche Meinungen, von »absolut alles« bis hin zu »Dokumentation ist überflüssig.« Gar nicht zu dokumentieren, ist allerdings eine unpopuläre Meinung, die kaum jemand ernsthaft vertritt. Man ist sich allgemein einig darüber, dass zumindest sämtliche Klassen sowie Methoden mit dem Access-Modifier `public` dokumentiert werden sollten. Meist werden auch `protected`-Methoden noch als unbedingt zu dokumentieren erwähnt. Methoden mit eingeschränkterer Sichtbarkeit zu dokumentieren, wird dagegen häufig als nicht notwendig angesehen, eine Meinung, die ich nur begrenzt teile, denn selbst der eigene Code ist nach einem halben Jahr nicht mehr offensichtlich. Javadoc an allen Methoden erleichtert es enorm, Code zu verstehen. Und wenn Sie eine übersichtlichere HTML-Dokumentation erzeugen wollen oder die Interna Ihres Codes nicht komplett preisgeben möchten, dann können Sie beim Generieren der Dokumentation den Schalter `-protected` verwenden.

Wirklich unnötig, darüber besteht wieder mehr Einigkeit, ist die Dokumentation von privaten Feldern, also Instanz- oder Klassenvariablen. Diese alle zu dokumentieren, bläht die Dokumentation auf und trägt meist wenig zum Verständnis bei. Wenn die

Bedeutung eines Feldes nicht offensichtlich ist, dann ist hier ein einfacher Codekommentar angebracht.

1.8 JARs erstellen und ausführen

Damit ist der Überblick über die wichtigsten mit dem JDK installierten Kommandozeilenwerkzeuge beinahe abgeschlossen, zuletzt bleibt noch, `jar` zu erwähnen. Dieses Werkzeug dient dem Umgang mit JAR-Dateien, einem Archivformat speziell für Java-Anwendungen und Bibliotheken. JARs sind im Wesentlichen ZIP-Dateien, die Java-Klassen enthalten. Sie lassen sich auch mit jedem Programm, das mit ZIP-Dateien umgehen kann, einsehen und verarbeiten. Genau wie im Dateisystem müssen die Klassen im Archiv in einer Verzeichnisstruktur liegen, die ihrem Package entspricht. Außerdem enthält ein korrektes JAR immer die Datei *META-INF/MANIFEST.MF*, sie enthält Metainformationen über die im JAR enthaltenen Klassen.

1.8.1 Die Datei »MANIFEST.MF«

Das Manifest einer JAR-Datei kann Informationen über die Klassen im JAR enthalten. Im einfachsten Fall muss es aber nicht einmal das, das einfachste Manifest deklariert nur, dass es sich um ein Manifest handelt:

```
Manifest-Version: 1.0
Created-By: 1.8.0-ea (Oracle Corporation)
```

Listing 1.7 Das einfachste Manifest

Wie Sie sehen, sind die Daten im Manifest als Schlüssel-Wert-Paare hinterlegt. Jede Zeile hat das Format `<Schlüssel>:<Wert>`. Die beiden Einträge in diesem Manifest sagen lediglich aus, dass es sich um ein Manifest der Version 1.0 handelt, das vom JDK 1.8.0 Early Access (*ea*) erstellt wurde.

Vorsicht beim Editieren von Manifesten

Manifeste sind empfindliche Dateien, die durch unvorsichtiges Editieren leicht zu beschädigen sind. Beachten Sie daher immer zwei grundlegende Probleme, wenn Sie ein Manifest bearbeiten:

1. Eine Zeile darf nie länger als 72 Byte sein. Das entspricht nicht immer 72 Zeichen, da viele Unicode-Zeichen mehr als ein Byte belegen. Solange Sie nur Zeichen aus dem ASCII-Zeichensatz verwenden, sind 72 Zeichen aber genau 72 Byte. Sollte eine Zeile länger sein, so müssen Sie nach 72 Zeichen einen Zeilenumbruch einfügen und die nächste Zeile mit einem einzelnen Leerzeichen beginnen, um sie als Fortsetzung zu markieren.

2. Jede Zeile des Manifests muss mit einem Zeilenumbruch abgeschlossen werden. Das bedeutet, dass die letzte Zeile des Manifests immer leer ist, da auch die letzte »echte« Zeile mit einem Zeilenumbruch endet. Fehlt der Zeilenumbruch, wird die Zeile ignoriert. Je nachdem, welcher Eintrag des Manifests betroffen ist, kann dies zu Fehlern führen, die nur sehr schwer zu finden sind.

Da das Editieren des Manifests so fehleranfällig ist, ist meistens davon abzuraten. Ein einfaches Manifest wird von `jar` erstellt, dazu mehr in Abschnitt 1.8.3, »JARs erzeugen«. Ein Projekt, das komplex genug ist, um spezielle Manifest-Einträge zu benötigen, kann normalerweise auch in anderer Hinsicht von einem Build-Tool (siehe Kasten »Kompilieren größerer Projekte« in Abschnitt 1.4.7) profitieren.

Es gibt neben diesen beiden Attributen noch viele weitere, die rein informativer Natur sind, es gibt aber auch Attribute, die technische Auswirkungen haben. Zwei davon seien hier besonders erwähnt.

Das Attribut »Class-Path«

Das Attribut `Class-Path` enthält eine Liste weiterer JARs, die vorhanden sein müssen, weil sie Klassen enthalten, die in diesem JAR verwendet werden. Dieses Attribut ist besonders wertvoll für Applets, da die in `Class-Path` aufgeführten JARs automatisch heruntergeladen werden. Jeder Eintrag wird dabei als URL relativ zur URL dieses JARs interpretiert. Mehrere Einträge werden durch Leerzeichen getrennt. Dieses Attribut überschreitet besonders häufig das Limit für die Zeilenlänge.

```
Manifest-Version: 1.0
Created-By: 1.8.0-ea (Oracle Corporation)
Class-Path: utils.jar otherjar.jar verzeichnis/jar.jar
```

Listing 1.8 Manifest mit »Class-Path«-Eintrag

Das Attribut »Main-Class«

JARs dienen nicht nur dem einfachen Transport von Java-Klassen, sie können auch eine Anwendung enthalten, die direkt aus dem JAR ausgeführt werden kann. Dies wird durch den Manifest-Eintrag `Main-Class` gesteuert.

```
Manifest-Version: 1.0
Created-By: 1.8.0-ea (Oracle Corporation)
Main-Class: de.kaiguenster.javaintro.reverse.Reverse
```

Listing 1.9 Manifest mit »Main-Class«-Eintrag

Das Attribut `Main-Class` enthält den voll qualifizierten Namen einer Klasse, die im JAR eingepackt ist. Diese Klasse muss eine `main`-Methode enthalten. Ein so präpariertes JAR kann ausgeführt werden, wie im nächsten Abschnitt beschrieben. Es ist zwar möglich, ein Java-Programm direkt aus dem JAR auszuführen, wenn `Main-Class` nicht gesetzt ist, der Benutzer muss dann aber den Namen der Klasse kennen, die die `main`-Methode enthält. Mit dem `Main-Class`-Attribut ist dieses Wissen nicht notwendig, der Name des JARs reicht aus.

1.8.2 JARs ausführen

Es gibt zwei Szenarien, ein JAR auszuführen: Entweder es hat einen `Main-Class`-Eintrag im Manifest oder nicht.

JAR mit Main-Class-Eintrag

Der einfachere Fall ist, dass im Manifest, wie oben beschrieben, eine `Main-Class` konfiguriert ist. In diesem Fall kann `java` mit dem Parameter `-jar` das Programm sofort ausführen:

```
java -jar meinjar.jar <Aufrufparameter>
```

Mit diesem Aufruf wird die `main`-Methode in der Hauptklasse von `meinjar.jar` ausgeführt. Die Aufrufparameter werden wie gewohnt übergeben. Wenn das Manifest das Attribut `Class-Path` enthält, werden die dort aufgeführten JARs dem Klassenpfad hinzugefügt und stehen dem Programm zur Verfügung.

JAR ohne Main-Class-Eintrag

Ist für das JAR keine Hauptklasse konfiguriert, muss ihr Klassenname bekannt sein. Beim Aufruf von Java wird das JAR dem Klassenpfad hinzugefügt, ansonsten ist der Aufruf derselbe wie bei einer nicht archivierten Klasse.

```
java -cp meinjar.jar package.Klasse <Aufrufparameter>
```

Der Parameter `-cp` steht für Klassenpfad (*class path*) und teilt der JVM mit, wo nach Klassen gesucht werden soll. Da die Klasse in der JAR-Datei liegt, muss diese dem Klassenpfad hinzugefügt werden. Ist im Manifest des JARs das Attribut `Class-Path` gesetzt, werden die dort aufgeführten Dateien auch in diesem Fall dem Klassenpfad hinzugefügt.

1.8.3 JARs erzeugen

Wenn Sie mit NetBeans arbeiten, wird automatisch ein JAR Ihres Projekts erzeugt und im Unterverzeichnis *dist* abgelegt. Aber Sie können ein JAR auch von der Kommandozeile aus erzeugen. Dazu rufen Sie im Ausgabeverzeichnis Ihres Projekts

```
jar -cvf meinjar.jar de
```

auf. Vorausgesetzt, das oberste Package Ihres Projekts heißt `de`, ansonsten setzen Sie den letzten Parameter entsprechend. Der erste Parameter enthält diverse Optionen für den `jar`-Befehl (siehe Tabelle 1.6).

Option	Bedeutung
`-c`	Es soll ein neues JAR erzeugt werden.
`-v`	Schreibt ausführliche Ausgaben auf die Konsole.
`-f`	Es wird eine JAR-Datei angegeben, die von `jar` erzeugt werden soll. Der Dateiname wird als weiterer Parameter übergeben.

Tabelle 1.6 Allgemeine »jar«-Optionen

Der `jar`-Befehl erzeugt automatisch ein Manifest und legt es an der richtigen Stelle im erstellten Archiv ab. Der Inhalt dieses Manifests entspricht dem oben gezeigten minimalen Manifest. Es gibt aber noch weitere Optionen für `jar`, die steuern, wie das Manifest erzeugt wird (siehe Tabelle 1.7).

Option	Bedeutung
`-e`	Es wird als zusätzlicher Parameter ein voll qualifizierter Klassenname übergeben, der als Wert des Attributs `Main-Class` ins Manifest geschrieben wird.
`-m`	Es wird als zusätzlicher Parameter der Name einer Manifest-Datei angegeben, deren Inhalt in das erzeugte Manifest übernommen wird.
`-M`	Es wird kein Manifest erzeugt.

Tabelle 1.7 Spezielle »jar«-Optionen

Leider folgt der Aufruf von `jar` nicht der Konvention, die Sie vielleicht von anderen Kommandozeilentools her kennen, vor allem aus der UNIX-Welt. Dort wäre es üblich, die Parameter `-f`, `-e` und `-m`, jeweils direkt gefolgt von ihrem Wert, zu verwenden. Der Aufruf sähe zum Beispiel so aus:

```
jar -cvf meinjar.jar -e de.kaiguenster.Beispiel de
```

Das ist aber **falsch**, `jar` versteht die Parameter so nicht. Die Optionen, die einen zusätzlichen Parameter erwarten, müssen alle in der Folge von Optionen stehen, die dazugehörigen Werte müssen danach in derselben Reihenfolge übergeben werden. Der korrekte Aufruf lautet also:

```
jar -cvfe meinjar.jar de.kaiguenster.Beispiel de
```


1.8.4 JARs einsehen und entpacken

Der `jar`-Befehl kann auch den Inhalt einer JAR-Datei auflisten oder die Datei entpacken. Dafür muss beim Aufruf lediglich anstelle von `-c` eine andere Operation angegeben werden. Zum Listen des Inhalts lautet die Option `-t`, zum Entpacken `-x`. Um den Inhalt einer JAR-Datei anzeigen zu lassen, lautet der Aufruf also:

```
jar -tf meinjar.jar
```

Zum Entpacken rufen Sie folgenden Befehl auf:

```
jar -xf meinjar.jar
```

In beiden Fällen können Sie aber auch ein grafisches Tool verwenden, das mit ZIP-Dateien umgeht, diese Lösung ist meist komfortabler.

1.9 Mit dem Debugger arbeiten

Es gibt noch ein weiteres Werkzeug, das die Entwicklung von und vor allem die Fehlersuche in Java-Programmen enorm erleichtert: den Debugger. Der Debugger selbst ist kein Bestandteil der JVM. Sie stellt aber eine Schnittstelle bereit, mit der sich andere Programme verbinden können, um diese Funktionalität zur Verfügung zu stellen. Alle Java-Entwicklungsumgebungen enthalten einen Debugger.

Ein Debugger wird dazu verwendet, in einem laufenden Java-Prozess Programmzeile für Programmzeile zu beobachten, was das Programm tut, welche Methoden es aufruft, welche Werte in seinen Variablen gespeichert sind und mehr. Der Debugger ist ein mächtiges Werkzeug.

1.9.1 Ein Programm im Debug-Modus starten

In jeder verbreiteten Entwicklungsumgebung gibt es eine einfache Möglichkeit, ein Programm im Debug-Modus zu starten, so auch in NetBeans. Aber es gibt vorher eine wichtige Einstellung zu prüfen: Stellen Sie in den Projekteigenschaften (zu finden unter File • Project Properties) in der Kategorie Compiling sicher, dass der Haken bei der Option Compile on Save **nicht** gesetzt ist (siehe Abbildung 1.8). Ist die Option aktiviert, so stehen einige Optionen beim Debugging nicht zur Verfügung.

Um ein Programm aus der Entwicklungsumgebung zu debuggen, müssen Sie nur anstelle des Knopfes Run Project den Knopf Debug Project rechts daneben benutzen. Wenn Sie die Funktion jetzt ausprobieren, werden Sie aber noch keinen Unterschied zur normalen Ausführung feststellen. Dazu benötigen Sie noch einen Breakpoint.

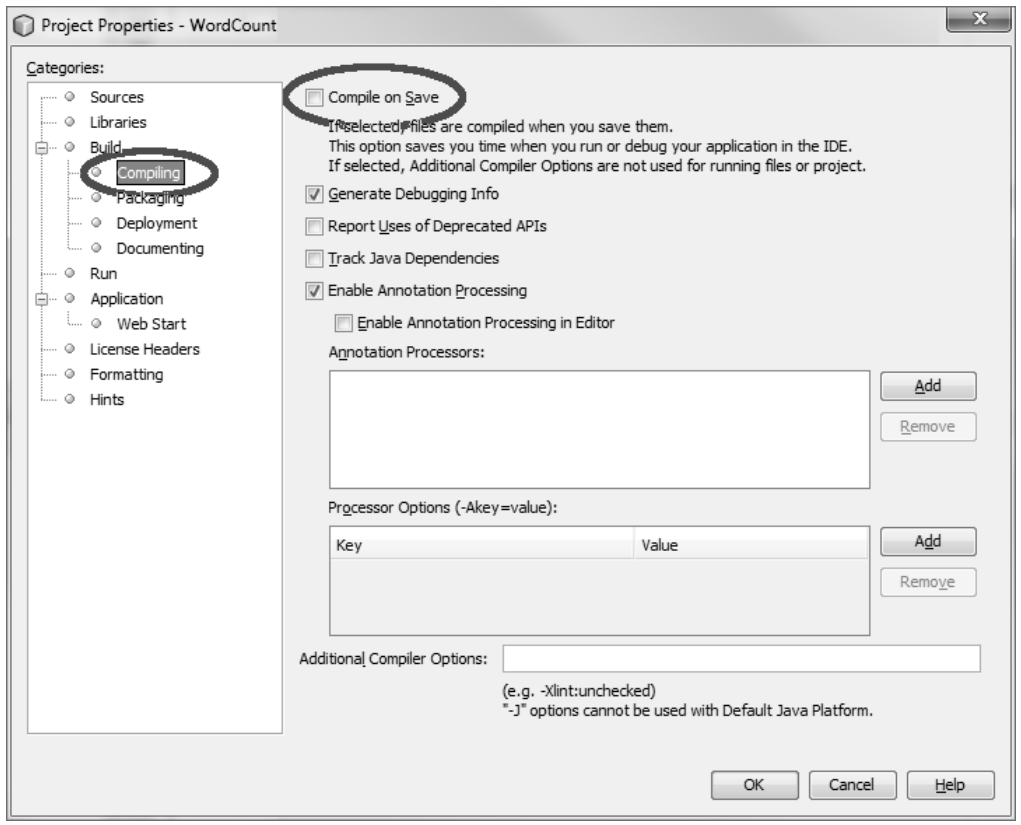

Abbildung 1.8 »Compile on Save« deaktiviert

1.9.2 Breakpoints und schrittweise Ausführung

Ein *Breakpoint* (Unterbrechungspunkt) ist eine Stelle im Code, an der die Ausführung des Programms angehalten wird. Sie setzen einen Breakpoint, indem Sie auf die Zeilennummer klicken, in der Sie das Programm anhalten möchten. Breakpoints können schon gesetzt werden, bevor Sie das Programm ausführen – gerade bei kurzen Programmen wie den Beispielen aus diesem Kapitel ein großer Vorteil.

Ist das Programm an einem Breakpoint angehalten, so können Sie es mit den Funktionen aus der Debug-Toolbar Schritt für Schritt ausführen und an jeder Stelle den aktuellen Zustand des Programms einsehen (siehe Abbildung 1.9), wie im nächsten Abschnitt gezeigt.

Abbildung 1.9 NetBeans Debug-Toolbar

Von links nach rechts haben die Knöpfe der Toolbar folgende Bedeutungen:

- Finish Debugger Session: Das Programm wird sofort beendet, der restliche Code wird nicht ausgeführt.
- Pause: Es werden alle Threads des Programms angehalten. Die ist eine Fortgeschrittenenfunktion, die an dieser Stelle noch keinen Nutzen hat.
- Continue: Das Programm wird weiter ausgeführt bis zum Programmende oder zum nächsten Breakpoint.
- Step Over: Die aktuelle Zeile des Programms, in NetBeans zu erkennen am grünen Hintergrund, wird ausgeführt. In der nächsten Zeile wird das Programm wieder angehalten.
- Step over Expression: Es wird der nächste Ausdruck ausgeführt. Dies ermöglicht eine feinere Kontrolle als die Funktion Step over, da eine Zeile mehrere Ausdrücke enthalten kann. Nehmen Sie zum Beispiel diese Zeile aus dem WordCount-Programm: `wordCounts.put(word, wordCounts.getOrDefault(word, 0) + 1)`. Der erste Aufruf von Step over Expression führt `wordCounts.getOrDefault` aus, der zweite `wordCounts.put`. Step Over würde beides gleichzeitig ausführen.
- Step Into: Ist der nächste Ausdruck ein Methodenaufruf, dann steigt Step Into in diese Methode ab und hält den Programmablauf in der ersten Zeile der Methode an.
- Step Out: Führt die aktuelle Methode bis zum Ende aus und kehrt in die aufrufende Methode zurück. Dort wird der Ablauf erneut angehalten.
- Run to Cursor: Führt das Programm bis zu der Zeile aus, in der der Cursor steht. Dort wird der Ablauf angehalten.
- Apply Code Changes: Seit Version 1.4.2 beherrscht Java das sogenannte *Hot Swapping*. Dadurch ist es möglich, Code in einem laufenden Programm durch die Debug-Schnittstelle zu ersetzen. So ist es nicht mehr nötig, das Programm neu zu kompilieren, um eine Änderung zu testen. Hot Swapping kann allerdings nicht alle Änderungen übernehmen.

1.9.3 Variablenwerte und Call Stack inspizieren

Wenn der Programmablauf angehalten ist, können Sie den Zustand des Programms im Detail betrachten. Zu diesem Zweck öffnet NetBeans, wenn der Debugger gestartet wird, zwei neue Ansichten.

Am unteren Fensterrand finden Sie die Ansicht Variables (siehe Abbildung 1.10). Hier können Sie die Werte sämtlicher Variablen einsehen, die an dieser Stelle im Programm zur Verfügung stehen.

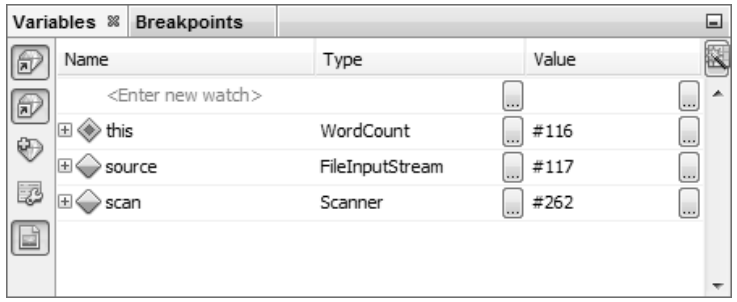

Abbildung 1.10 Die Variablenansicht des Debuggers

In der linken Spalte sehen Sie den Namen der Variablen, daneben ihren Typ und schließlich, für ausgewählte Typen wie Strings und Zahlen, ihren Wert. Mit dem Pluszeichen neben dem Variablennamen können Sie die »inneren Werte« eines Objekts inspizieren, ein Klick darauf öffnet die Sicht auf alle Instanzvariablen des Objekts.

Im Screenshot und bei Verwendung des Debuggers fällt eine Variable namens `this` auf, die nirgends deklariert wird, aber trotzdem fast immer vorhanden ist. `this` ist keine echte Variable, sondern ein Schlüsselwort, über das Sie immer auf das Objekt zugreifen können, in dem sich die aktuelle Methode befindet, und zwar nicht nur im Debugger, sondern auch im Code. Wofür das gut ist, wird in Kapitel 5, »Klassen und Objekte«, ein Thema werden. Für den Moment können Sie darüber im Debugger auf Instanzvariablen zugreifen.

Die zweite wichtige Ansicht des Debuggers befindet sich am linken Fensterrand unter dem Titel Debugging. Hier können Sie den Call Stack (zu Deutsch: Aufrufstapel) einsehen (siehe Abbildung 1.11).

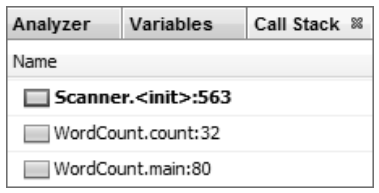

Abbildung 1.11 Die Stackansicht des Debuggers

Der Call Stack ist der Pfad von Methoden, der zur aktuellen Stelle im Programm führt. Wird eine Methode aufgerufen, so wird sie auf den Stack gelegt. Ruft diese Methode nun eine weitere Methode auf, so wird die neue Methode auf dem Stack über ihren Aufrufer gelegt. Die aufrufende Methode mit allen ihren Variablen bleibt unverändert auf dem Stack liegen, sie wird lediglich vorübergehend durch die neue Methode verdeckt. Endet die oberste Methode auf dem Stack, kehrt die Ausführung sofort zur nächsten Methode zurück und setzt diese an der Stelle fort, wo der Methodenaufruf

erfolgte. Durch den Call Stack weiß Java überhaupt, wo die Ausführung fortzusetzen ist, nachdem eine Methode endet.

Schauen Sie sich beispielsweise den Screenshot in Abbildung 1.11 an. Als unterste Methode auf dem Stack sehen Sie die `main`-Methode der Klasse `WordCount`. Die Zahl nach dem Methodennamen gibt an, in welcher Zeile der Datei die Ausführung gerade steht. Von der `main`-Methode wurde die Methode `count` der Klasse `WordCount` gerufen. Diese wiederum ruft den Konstruktor der Klasse `Scanner` auf, der zurzeit oben auf dem Stack liegt. In der Stackansicht werden Konstruktoren immer als `<init>` dargestellt.

Durch einen Doppelklick auf einen Eintrag können Sie zu einer anderen Methode auf dem Stack springen und die Variablenwerte in dieser Methode einsehen.

1.9.4 Übung: Der Debugger

Damit kennen Sie nun die wichtigsten Funktionen des Debuggers. Das gibt Ihnen die Möglichkeit, den Ablauf eines Java-Programms im Detail nachzuvollziehen. Dies sollen Sie jetzt an zwei Beispielen aus diesem Kapitel ausnutzen, um zum einen das Arbeiten mit dem Debugger kennenzulernen und zum anderen ein Gefühl für den Ablauf eines Programms zu entwickeln, bevor Sie dann in Kapitel 3 selbst anfangen zu programmieren.

Reverse

Öffnen Sie zunächst das Projekt Reverse und dort die Klasse `Reverse`. Tragen Sie in der Aufrufkonfiguration (der Punkt Customize im Dropdown-Menü neben dem Run-Knopf) die Zeichenkette »Hallo, Welt!« als Aufrufparameter im Feld Arguments ein.

Setzen Sie einen Breakpoint in der ersten Zeile der `main`-Methode. Führen Sie das Programm im Debug-Modus aus, indem Sie den Knopf Debug Project betätigen. Gehen Sie das Programm mit Step Over in Einzelschritten durch. Wenn Sie den Aufruf der Methode `reverse` erreichen, folgen Sie diesem Aufruf mit Step Into. Beobachten Sie, welche Zeilen ausgeführt und welche übersprungen werden und wie sich die Werte der Variablen verändern. Achten Sie besonders auf diese Punkte:

- Der Rumpf des `if`-Statements `if (args.length < 1)` wird nicht ausgeführt. Das liegt natürlich daran, dass seine Bedingung nicht erfüllt ist.
- Beobachten Sie die Variablen, während die `for`-Schleife in der `main`-Methode ausgeführt wird. Die Zählvariable `i` wird nach jedem Durchlauf um 1 erhöht. Die Variable `parameter` wird mit den Inhalten von `args` befüllt.
- Beobachten Sie auch die `for`-Schleife in der Methode `reverse`. Achten Sie auch hier auf die Zählvariable und darauf, wie `out` Zeichen für Zeichen wächst.

Fibonacci

Öffnen Sie nun die Klasse `Fibonacci` im gleichnamigen Projekt (siehe Abschnitt 1.5.1). Setzen Sie einen Breakpoint in der ersten Zeile der Methode `fibonacci`, und starten Sie das Programm im Debug-Modus mit dem Aufrufparameter 4. Verfolgen Sie auch dieses Programm Schritt für Schritt, folgen Sie jedem Aufruf von `fibonacci` mit STEP INTO, und beobachten Sie dabei den Call Stack auf der linken Seite des Fensters.

Sie sehen dort, dass mit jedem Aufruf von `fibonacci` die Methode wieder auf dem Stack hinzugefügt wird und dort bis zu viermal vorkommen kann. Wechseln Sie durch Doppelklick in die verschiedenen Aufrufe der Methode, und achten Sie auf die Variablen. Sie werden feststellen, dass die Variablenwerte in jedem Aufruf unterschiedlich sind. Die verschiedenen Aufrufe haben ihre eigenen Variablen und sind auch ansonsten völlig unabhängig voneinander.

Vollziehen Sie anhand des Debuggers nach, mit welchen Parametern die Methode von wo gerufen wird. Zeichnen Sie es zur besseren Übersicht auf. Da `fibonacci` sich selbst zweimal rekursiv aufruft, sollte Ihre Skizze einem Baum ähnlich sehen (siehe Abbildung 1.12).

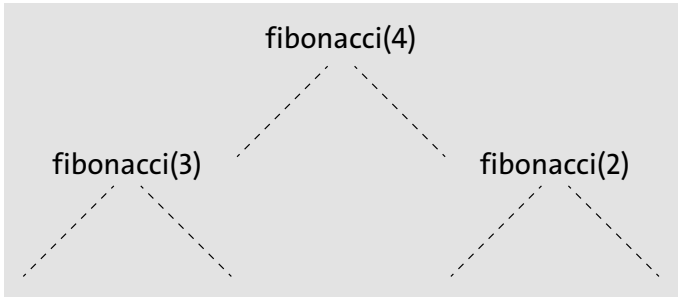

Abbildung 1.12 Rekursive Aufrufe von »fibonacci«

Spätestens durch das Diagramm werden Sie bemerkt haben, dass der Algorithmus einige Fibonacci-Zahlen mehrfach berechnet, um zu seinem endgültigen Ergebnis zu gelangen. Es ist nicht weiter schwierig, dieses Problem zu beheben, indem Sie einmal berechnete Werte in einem Array speichern und sie beim zweiten Mal von dort lesen, anstatt sie erneut zu berechnen. Dadurch würde aber der Algorithmus weniger klar und verständlich.

1.10 Das erste eigene Projekt

Damit kann es nun auch fast losgehen, es fehlt nur noch ein eigenes Projekt, um Ihre ersten Programmiererfolge zu verwirklichen. Wählen Sie dazu in NetBeans den Menüpunkt FILE • NEW PROJECT.

Der Dialog, in dem Sie ein neues Projekt anlegen, bietet eine Vielzahl von Projekttypen, Vorlagen, aus denen ein Projekt erzeugt werden kann. Für den Anfang ist davon aber nur der Typ JAVA APPLICATION aus dem Ordner JAVA interessant. Wählen Sie ihn aus, und klicken Sie auf NEXT (siehe Abbildung 1.13).

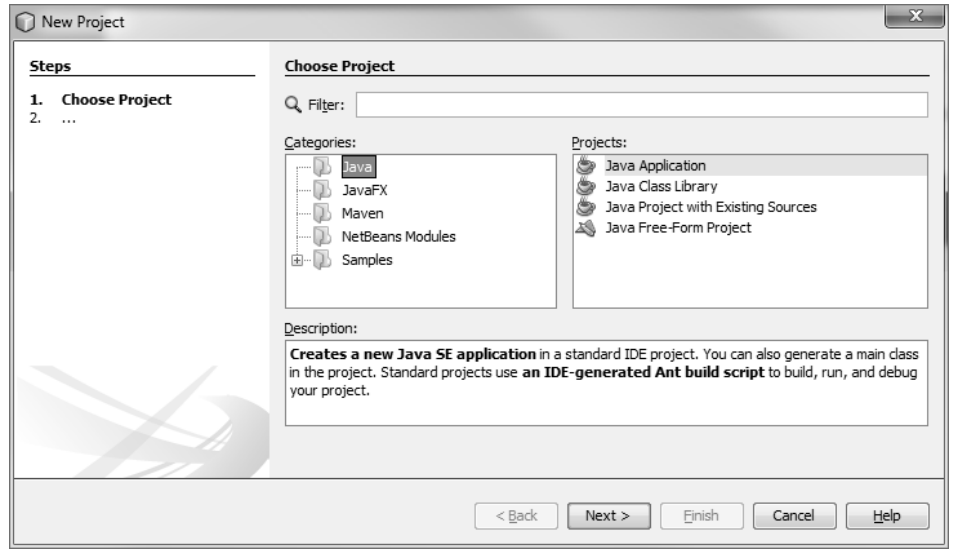

Abbildung 1.13 Neues Projekt anlegen – Schritt 1

Auf der zweiten Seite des Dialogs geben Sie Ihrem Projekt einen Namen, wählen einen Speicherort aus und geben den voll qualifizierten Namen Ihrer Hauptklasse an (siehe Abbildung 1.14). Fertig.

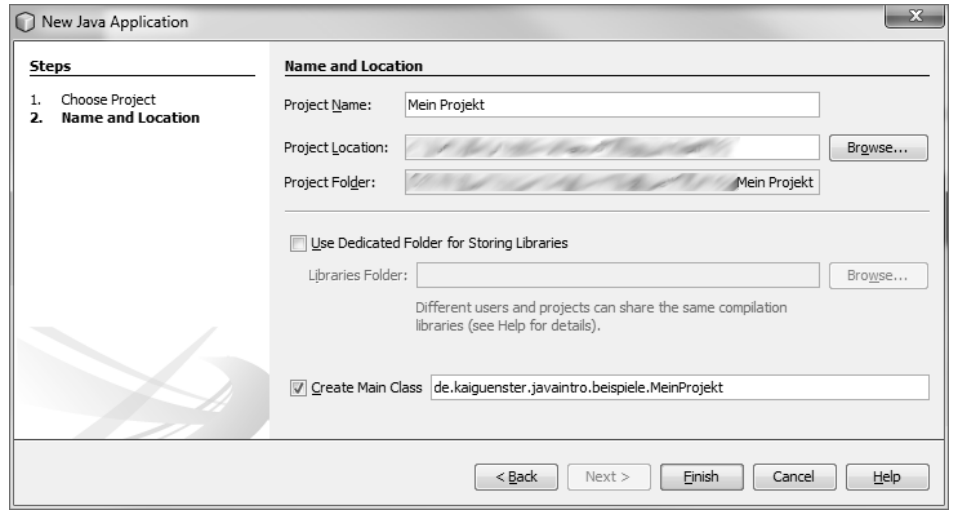

Abbildung 1.14 Neues Projekt anlegen – Schritt 2

Wenn Sie die Beispiele und Übungen der nächsten Kapitel angehen, haben Sie die Wahl, ob Sie für jedes Programm ein neues Projekt anlegen oder alle Programme in einem Projekt bündeln. Es kann in einem Projekt mehrere Klassen mit `main`-Methode geben, ohne dass es deswegen zu Problemen kommt. Beide Ansätze haben aber kleine Vor- und Nachteile.

Wenn Sie alle Programme in einem Projekt bündeln, hat das den Nachteil, dass beim Kompilieren des Projekts mehr Klassen kompiliert werden, als Sie eigentlich gerade benötigen. Dafür können Sie Methoden, die Sie bereits für eine Übung entwickelt haben, später für andere Übungen wiederverwenden. Wenn Sie diesen Ansatz wählen, legen Sie bitte im Projekt ein eigenes Package je Übung oder Beispiel an.

Für jede Übung ein eigenes Projekt anzulegen, minimiert die Zeit, die zum Kompilieren benötigt wird. Aber um auf fertigen Code aus anderen Übungen zuzugreifen, müssen Sie entweder Code kopieren oder Projektabhängigkeiten einrichten.

1.11 Zusammenfassung

In diesem Kapitel haben Sie einen umfassenden Überblick über die Welt von Java erhalten. Sie kennen die Bestandteile der Java-Plattform, die wichtigsten Kommandozeilenwerkzeuge und die wichtigsten Packages der Klassenbibliothek. Sie haben den grundlegenden Umgang mit der Entwicklungsumgebung NetBeans und mit dem Debugger kennengelernt. Sie wissen, wie Sie Java-Programme ausführen und wie Sie Javadoc schreiben und daraus die HTML-Dokumentation generieren. Vor allem haben Sie an ersten Beispielen gesehen, wie ein Algorithmus aufgebaut wird und wie Sie vom abstrakten Algorithmus zu einer Implementierung in Java gelangen. Als Nächstes werden Sie lernen, mit Zahlen und Zeichenketten zu arbeiten, Variablen zu verwenden und Berechnungen durchzuführen.

Kapitel 12

Dateien, Streams und Reader

Fast jedes Programm, das über eine einfache Übung hinausgeht, braucht eine Art von Persistenz. Sie wollen eine Sitzung mit dem Programm unterbrechen können, das Programm zu einem späteren Zeitpunkt neu starten und genau da weiterarbeiten, wo Sie zuletzt aufgehört haben. Für Programme auf Ihrem Arbeitsplatzrechner bedeutet das fast immer, dass Dateien geschrieben und gelesen werden.

Kaum ein Programm kommt ohne Ein- und Ausgabe von Dateien aus. Sie wollen eine Textdatei, die Sie bearbeiten, speichern und später wieder einlesen können. Dasselbe gilt für Spielstände in fast jedem Spiel. Für einen Musicplayer müssen Sie die Musikdaten einlesen usw. Sie kommen um File I/O, also den Umgang mit Dateien, einfach nicht herum. Zumindest nicht bei Anwendungen, die auf Ihrem Arbeitsplatzrechner ausgeführt werden. Für Anwendungen, die auf einem Server betrieben werden, kommen häufiger Datenbanken zum Einsatz, die uns aber in diesem Kapitel nicht beschäftigen sollen.

Java bietet für Ein- und Ausgabeoperationen zwei verschiedene APIs an: das klassische *Blocking I/O* aus dem `java.io`-Package und das *Non-Blocking I/O* oder `new I/O` aus `java.nio`. In diesem Buch soll es nur um Blocking I/O gehen, denn die Non-Blocking I/O ist viel komplexer und schwieriger in der Benutzung, ihre Vorteile kommen aber in Arbeitsplatzanwendungen so gut wie nie zum Tragen.

»java.io« und »java.nio« – was ist der Unterschied?

Der Unterschied zwischen `java.io` und `java.nio` ist, dass Operationen der einen API blockierend (*blocking*) und Operationen der anderen nicht blockierend (*non-blocking*) sind. Das bedeutet einfach, dass blockierende Operationen auf ihr Ergebnis warten. Wenn Sie mit den Methoden aus dem klassischen `java.io`-Package aus einer Netzwerkverbindung lesen, dann wartet der entsprechende Methodenaufruf so lange, bis entweder Daten zur Verfügung stehen oder bis die Verbindung geschlossen wird. Während dieser Zeit kann der Thread, in dem diese Operation wartet, nichts anderes tun. Für eine Anwendung, die Sie auf Ihrem Rechner ausführen, ist das selten ein Problem. Sie können die I/O-Operationen wenn nötig in einem eigenen Thread ausführen, so dass der Rest Ihres Programmes nicht blockiert wird.

Dies ist aber keine Möglichkeit für ein Serverprogramm, das Tausende oder mehr Netzwerkverbindungen gleichzeitig verarbeiten muss. Wenn Sie für jede dieser Verbindungen einen eigenen Thread starten wollen, hat das sehr schlechte Auswirkungen auf Speicherverbrauch und Geschwindigkeit Ihres Programms. Genau für diese Fälle wurde `java.nio` geschaffen. Mit dieser neueren API blockieren Ein-/Ausgabeoperationen den Thread nicht mehr; so können Sie viele Verbindungen in einem Thread verarbeiten.

12.1 Dateien und Verzeichnisse

Dateioperationen mit `java.io` werden in Java immer, direkt oder indirekt, durch ein Objekt des Typs `java.io.File` abgebildet. Dabei kann `File` aber nicht selbst aus Dateien lesen oder in sie schreiben, dazu benötigen Sie einen `Reader` oder `Writer` (für Textdateien) bzw. einen `InputStream` oder `OutputStream` (für Binärdateien). Aber viele andere Operationen und Informationen über eine Datei finden Sie im `File`-Objekt.

12.1.1 Dateien und Pfade

Dieses Objekt ist eine objektorientierte Darstellung eines Pfades, und Sie erzeugen ein File auch immer aus einer Pfadangabe, entweder absolut oder relativ. Ein absoluter Pfad geht von einem Wurzelverzeichnis aus, zum Beispiel *C:* unter Windows oder */* unter Linux. Ein relativer Pfad bezieht sich dagegen auf das aktuelle Verzeichnis des Benutzers, normalerweise das Verzeichnis, aus dem er Ihr Programm aufgerufen hat.

```
File windowsDatei = new File("C:\\home\\kguenster\\text.txt");
File linuxDatei = new File("/home/kguenster/text.txt");
```

Listing 12.1 Absolute »File«-Objekte unter Windows und Linux erzeugen

So einfach sind Files zu erzeugen. Wenn die Datei mit diesem Pfad nicht existiert, wird sie durch das Erzeugen des `File`-Objekts auch nicht angelegt.

Aber Sie sehen auch sofort ein Problem: Pfade werden für verschiedene Betriebssysteme unterschiedlich angegeben. So verwendet Windows den Backslash, um zwei Verzeichnisse im Pfad zu trennen (achten Sie in String-Konstanten immer darauf, den Backslash zu doppeln!), Linux den normalen Slash. Außerdem gibt es in Windows Laufwerksbuchstaben, ein Konzept, das Linux völlig fremd ist.

Beides macht keine Probleme, wenn Sie mit Pfaden arbeiten, die ein Benutzer eingegeben hat. Solange er das im für sein Betriebssystem richtigen Format tut, können

Sie die Eingabe an den `File`-Konstruktor weitergeben, und sie wird richtig verarbeitet. Wenn Sie aber programmatisch Dateipfade erzeugen, dann müssen Sie auf diese Details selbst achten. Das richtige Zeichen zum Trennen von Verzeichnissen in einer Pfadangabe finden Sie in der Konstanten `File.separator`. Sie können damit einen Pfad systemunabhängig zusammensetzen:

```
File datei = new File(File.separator + "home"
    + File.separator + "kguenster"
    + File.separator + "text.txt");
```

Listing 12.2 »File«-Objekt systemunabhängig erzeugen

Dieser Code funktioniert unter Linux immer, aber unter Windows bleibt das Problem der Laufwerksbuchstaben. Wie gezeigt, wird die Datei */home/kguenster/text.txt* auf dem aktuellen Laufwerk gefunden. Da es Laufwerksbuchstaben überhaupt nur unter Windows gibt, kann dieses Konzept nicht völlig systemunabhängig dargestellt werden. Sie können aber, unabhängig vom Betriebssystem, alle Wurzelverzeichnisse auflisten; dazu kennt `File` die statische Methode `listRoots`:

```
public File waehleWurzel(){
    File[] wurzeln = File.listRoots();
    if (wurzeln.length == 1){
        return wurzeln[0];
    } else {
        System. out.println("Bitte wählen Sie eine Wurzel");
        for (int i = 0; i < wurzeln.length; i++){
            System.out.println(i + ": " + wurzeln[i]);
        }
        int index = liesZahl();
        return wurzeln[index];
    }
}
```

Listing 12.3 Wurzelverzeichnis auswählen, ganz systemunabhängig

Unter allen Unix-artigen Betriebssystemen, also Linux, BSD, macOS und mehr, hat das von `listRoots` zurückgegebene Array nur einen Eintrag, nämlich */*. Nur unter Windows kann es mehrere Einträge geben, nämlich *C:*, *D:* usw. In diesem Fall wird der Benutzer gebeten, eine Wurzel auszuwählen. Anschließend können Sie ein neues `File`-Objekt relativ zur ausgewählten Wurzel erzeugen, indem Sie sie im Konstruktor angeben. Das funktioniert nicht nur mit Wurzeln, sondern mit allen Verzeichnissen – übergeben Sie sie an den Konstruktor eines `File`-Objekts, dann wird dessen Pfad relativ zum übergebenen Pfad aufgelöst.

```
File wurzel = waehleWurzel();
File datei = new File(wurzel, "home"
 + File.separator + "kguenster"
 + File.separator + "text.txt");
```

Listing 12.4 Dateien relativ zu anderen Dateien

Ein `File` gibt allerhand interessante Informationen über die Datei preis, die es repräsentiert. Vor allem, ob die Datei überhaupt existiert. Da ein `File` lediglich die objektorientierte Repräsentation eines Pfades ist, können Sie auch Files erzeugen, zu denen keine Datei existiert. Mit der Methode `exists` finden Sie heraus, ob es diese Datei gibt. Falls nicht, können Sie mit `createNewFile` eine Datei an der vom Pfad angegebenen Stelle anlegen oder mit `mkdir` ein Verzeichnis. Auch auf alle weiteren Informationen, die über eine Datei interessant sein könnten, haben Sie Zugriff durch die `File`-Methoden (siehe Tabelle 12.1).

Methode	Beschreibung
`isFile()`	Handelt es sich bei diesem `File`-Objekt um eine Datei (im Gegensatz zu einem Verzeichnis)?
`isDirectory()`	Handelt es sich um ein Verzeichnis?
`canRead()`	Hat der Benutzer Leserechte? (Mit dieser Methode gibt es ein Problem unter manchen Windows-Versionen: Es kann vorkommen, dass sie `true` zurückgibt, aber beim Zugriff trotzdem eine `AccessDeniedException` auftritt.)
`canWrite()`	Hat der Benutzer Schreibrechte?
`canExecute()`	Hat der Benutzer Ausführrechte?
`getName()`	Liefert den Namen der Datei, ohne vorangehende Pfadangabe.
`getParent()`, `getParentFile()`	Liefert das übergeordnete Verzeichnis, entweder als `String` mit `getParent` oder als `File`-Objekt mit `getParentFile`.
`lastModified()`	Liefert das letzte Änderungsdatum der Datei als `long`.
`length()`	Liefert die Größe der Datei in Byte als `long`.

Tabelle 12.1 Informationen über Dateien

Darüber hinaus können Sie Dateien manipulieren: `delete` löscht eine Datei, wenn Sie die nötigen Berechtigungen haben, `renameTo` benennt eine Datei um. Es gibt in `java.io.File` nach wie vor keine Methoden, die Dateien kopieren oder verschieben.

Verschieben können Sie Dateien zwar auf manchen Systemen mit `renameTo`, es gibt dafür aber keine Garantie. So waren beide Operationen immer schmerzhaft selbst zu implementieren, indem man aus der Quelldatei liest und in die Zieldatei schreibt. Seit Java 7 gibt es aber endlich eine Hilfsklasse, die diese Operationen bereitstellt.

12.1.2 Dateioperationen aus »Files«

Die Klasse `Files` ist eine Sammlung von Hilfsmethoden für alles, was mit Dateien zu tun hat. Aus unbekannten Gründen gibt es diese praktische Klasse aber nicht im `java.io`-Package, sondern nur in `java.nio.files`, damit ist es die einzige Klasse aus der Non-Blocking-I/O-API, die Sie auch beim alltäglichen Umgang mit Dateien regelmäßig benutzen werden.

Der große Nachteil dabei, die Hilfsklasse einer anderen API zu benutzen, ist, dass `Files` nicht mit `java.io.File` arbeitet, sondern mit `java.nio.file.Path`. Sie müssen also bei jeder Operation die Parameter von `File` nach `Path` konvertieren und die Rückgabewerte, falls Dateien zurückgegeben werden, wieder von `Path` nach `File` (nicht alle `Path`-Objekte lassen sich nach `File` konvertieren, aber für solche, die aus einer Operation auf einem `File` resultieren, ist es immer möglich).

In `Files` finden Sie unter anderem Methoden, die eine Datei verschieben oder kopieren können. Weitere nützliche Methoden werden Sie im Laufe des Kapitels kennenlernen, wenn es thematisch passend ist.

```
//File nach Path konvertieren
Path quellPath = quelle.toPath();
Path zielPath = ziel.toPath();
//ENTWEDER Datei kopieren
Path ergebnisPath = Files.copy(quellPath, zielPath);
//ODER Datei verschieben
Path ergebnisPath = Files.move(quellPath, zielPath);
//Ergebnis - eigentlich wieder das Ziel - nach File konvertieren
File ergebnis = ergebnisPath.toFile();
```

Listing 12.5 Dateien kopieren und verschieben mit »Files«

Solche Kopier- und Verschiebeoperationen sind nicht nur für Sie als Programmierer praktischer, sie sind auch schneller, als wenn Sie sie in Java selbst umsetzen, da das JDK dafür effizientere Systemaufrufe verwenden kann.

12.1.3 Übung: Dateien kopieren

Für Ihren ersten Kontakt mit Dateien eine einfache Aufgabe: Schreiben Sie ein Programm, das eine Datei kopiert. Es soll einen Quellpfad und einen Zielpfad als Aufruf-

parameter erhalten. Beide Pfade können absolut oder relativ sein. Der Quellpfad muss auf eine existierende Datei verweisen.

Wenn der Zielpfad auf ein bestehendes Verzeichnis verweist, dann soll die Datei in dieses Verzeichnis kopiert werden und ihren Namen beibehalten.

Verweist der Zielpfad auf eine bereits existierende Datei, dann soll ein Fehler ausgegeben werden.

Verweist der Zielpfad auf eine noch nicht existierende Datei, dann ist der Name dieses Pfades der neue Dateiname, unter dem die Datei angelegt wird. Existieren ein oder mehrere übergeordnete Verzeichnisse noch nicht, dann sollen auch sie angelegt werden.

Prüfen Sie weitere mögliche Fehler, und geben Sie sprechende Fehlermeldungen aus. Im Fehlerfall können Sie das Programm mit `System.exit(1)` sofort beenden. Die 1 ist ein Fehlercode, jede Zahl außer 0 bedeutet für `System.exit`, dass das Programm mit einem Fehler beendet wurde, nur `System.exit(0)` beendet das Programm ohne Fehler. Diese Werte können von Ihrem Betriebssystem ausgewertet werden. Die Lösung zu dieser Übung finden Sie im Anhang.

12.1.4 Verzeichnisse

Über ein Verzeichnis gibt es natürlich noch eine weitere interessante Information: seinen Inhalt. Um diesen zu ermitteln, gibt es an `File` die überladene Methode `listFiles`. Ohne Parameter gibt sie alle im Verzeichnis enthaltenen Dateien zurück.

Wenn Sie nur an bestimmten Dateien interessiert sind, dann können Sie an `listFiles` entweder einen `FileFilter` oder einen `FilenameFilter` übergeben. Die beiden Filterklassen unterscheiden sich nur darin, dass `FileFilter` das `File`-Objekt der gefundenen Datei zur Prüfung erhält, `FilenameFilter` den Dateinamen als String und das aktuelle Verzeichnis. Beide Filter sind funktionale Interfaces und können deshalb auch als Lambdas angegeben werden.

```
//Alle Dateien auflisten
File[] alleDateien = verzeichnis.listFiles();
//Alle Dateien mit der Endung .txt auflisten
File[] textDateien = verzeichnis.listFiles((parent, name) ->
 name.endsWith(".txt"));
//Alle Unterverzeichnisse auflisten
File[] unterverzeichnisse = verzeichnis.listFiles(file ->
 file.isDirectory());
```

Listing 12.6 Dateien in einem Verzeichnis auflisten

Auch zum Auflisten des Verzeichnisinhalts hat die Klasse `Files` Hilfsmethoden. Davon ist `list` die langweiligere, sie tut nichts anderes, als den Inhalt eines Verzeichnisses als einen `Stream` von `Path`-Objekten zurückzugeben. Interessanter ist da schon `walk`, das nicht nur den Inhalt des übergebenen Verzeichnisses auflistet, sondern auch den aller Unterverzeichnisse.

```
Files.walk(quelle.toPath()).forEach(System.out::println);
```

Listing 12.7 Den Inhalt eines Verzeichnisses und aller Unterverzeichnisse ausgeben

Optional können Sie auch angeben, dass nicht beliebig weit in Unterverzeichnisse abgestiegen wird, sondern nur bis zu einer bestimmten Tiefe. `walk(quelle, 1)` enthält nur den Inhalt des Verzeichnisses selbst, tut also dasselbe wie `list`; `walk(quelle, 2)` enthält den Inhalt dieses Verzeichnisses und seiner direkten Unterverzeichnisse usw.

Als Letztes haben Sie mit `Files.find` die Möglichkeit, in einem Verzeichnis und seinen Unterverzeichnissen nach Dateien zu suchen, die bestimmten Vorgaben entsprechen. Leider kommt auch hier wieder durch, dass `java.io` und `java.nio` nicht aus einem Guss stammen. Sie geben also die Suchkriterien nicht als `FileFilter` an, sondern als `BiPredicate`, das als Parameter das `Path`-Objekt der Datei und ein Objekt vom Typ `BasicFileAttributes` erhält, in dem sich Informationen wie Dateigröße und letzte Zugriffszeit finden.

```
File quelle = new File("E:\\media");
Files.find(quelle.toPath(), 20,
           (pfad, attr) -> attr.isRegularFile()
                        && attr.size() > 500l * 1024 * 1024)
     .forEach(System.out::println);
```

Listing 12.8 Große Dateien finden

So finde ich zum Beispiel Dateien, die zu viel Platz auf meiner Festplatte belegen. Es werden alle Dateien aufgelistet, die größer als 500 MB sind. `walk` und `find` sind sehr praktische Methoden, sie haben aber einen nicht unerheblichen Nachteil: Wenn sie auf ein Verzeichnis stoßen, auf das sie nicht zugreifen können, brechen sie mit einer Fehlermeldung ab. Es gibt mit diesen Methoden keine Möglichkeit, unlesbare Verzeichnisse zu ignorieren, weswegen Sie häufig doch auf `File.listFiles` zurückgreifen müssen.

12.1.5 Übung: Musik finden

Schreiben Sie eine Klasse `Musikfinder`, die von einem angegebenen Startverzeichnis aus alle Unterverzeichnisse durchsucht und alle MP3-Dateien findet, also solche Da-

teien, die die Endung *.mp3* haben. Als Consumer soll dem Musikfinder mitgegeben werden, was er mit den gefundenen Dateien tun soll.

Aufgrund der Problematik mit lesegeschützten Verzeichnissen verwenden Sie bitte nicht die Hilfsmethoden aus `Files`, sondern `File.listFile` und realisieren die Rekursion in die Unterverzeichnisse selbst.

Schreiben Sie eine `main`-Methode, die das Startverzeichnis als Aufrufparameter erwartet und mit dem `Musikfinder` den Pfad aller gefundenen Dateien ausgibt. Die Lösung zu dieser Übung finden Sie im Anhang.

12.2 Reader, Writer und die »anderen« Streams

Nachdem Sie jetzt Dateien finden und im Dateisystem navigieren können, ist der logische nächste Schritt, ihren Inhalt zu lesen und eigene Dateien zu schreiben. Diese Vorgänge sind in Java, wie auch alle anderen Ein- und Ausgabeoperationen, streambasiert. Diese Streams haben aber nichts mit der Stream-API aus dem vorigen Kapitel zu tun – das ist vielleicht der einzige Nachteil der neuen Streams, sie sorgen für Namensverwirrung mit den alten Klassen `InputStream` und `OutputStream`.

Streambasierte I/O bedeutet einfach nur, dass Sie nicht alle Daten im Speicher haben müssen, um mit ihnen zu arbeiten; Sie müssen zum Beispiel nicht den ganzen Inhalt einer Datei lesen, bevor Sie ihn verarbeiten können, und Sie müssen bei einer Netzwerkverbindung nicht darauf warten, dass alle Daten bei Ihnen eingegangen sind. Stattdessen lesen Sie Daten Stück für Stück aus einem `InputStream` ein, verarbeiten sie und können sie im Idealfall wieder aus dem Speicher entfernen, bevor Sie das nächste Stück lesen. Ebenso können Sie Daten in einen `OutputStream` schreiben, sobald sie zur Verfügung stehen, und müssen nicht erst warten, bis alle zu schreibenden Daten bereit sind.

Dabei macht es in Java einen Unterschied, ob Sie mit Textdaten oder mit Binärdaten arbeiten. Textdaten werden mit einem `Reader` gelesen und mit einem `Writer` geschrieben, für Binärdateien gibt es dafür `InputStream` und `OutputStream`.

Zunächst geht es um Ein- und Ausgabe von und in Dateien, aber dieselben Klassen werden bei allen I/O-Operationen verwendet. Später in diesem Kapitel werden Sie sehen, wie Sie mit denselben Klassen über ein Netzwerk kommunizieren. Die verschiedenen möglichen Datenquellen auf diese Art und Weise zu abstrahieren, erspart es Ihnen nicht nur, mehrere verschiedene APIs zu lernen, sondern hat auch den noch größeren Vorteil, dass Sie Methoden schreiben können, die bezüglich ihrer Datenquelle agnostisch sind. Wenn eine Methode einen `InputStream` als Parameter erhält und aus ihm liest, dann ist es innerhalb der Methode egal, ob die Daten aus einer Datei, einer Netzwerkverbindung oder doch nur aus einem `byte[]` im Speicher stammen.

12.2.1 Lesen und Schreiben von Textdaten

Es gibt in Java zwei unterschiedliche Klassenhierarchien für den Umgang mit Textdaten und mit Binärdaten. Textdaten werden aus einem `Reader` gelesen und in einen `Writer` geschrieben, für Binärdateien werden diese Funktionen von `InputStream` und `OutputStream` bereitgestellt, dazu mehr in Abschnitt 12.2.3.

Lesen aus einem Reader

Sie haben im vorigen Kapitel bereits gesehen, wie Sie mit einem `BufferedReader` Textdaten zeilenweise aus einer Datei lesen.

```
try (BufferedReader reader = new BufferedReader(new FileReader(dateiname))){
    …
}
```

Listing 12.9 Daten zeilenweise lesen

Das ist aber bereits eine spezialisierte Funktion, die nur `BufferedReader` bietet. Andere `Reader`, zum Beispiel der `FileReader`, wissen nichts von Zeilen, sie arbeiten nur mit Zeichen. Dazu bietet Reader eine parameterlose Methode `read`, die genau ein Zeichen liest. Das ist zwar die für den Entwickler einfachste Variante, ist aber auch äußerst ineffektiv. Der komplexere, aber bessere Weg, aus einem `Reader` zu lesen, ist, ein `char[]` als Puffer zu benutzen:

```
File quelle = new File(…);
char[] buffer = new char[1024];
try (Reader reader = new FileReader(quelle)) {
    int gelesen;
    while ((gelesen = reader.read(buffer)) > -1) {
        char[] geleseneDaten = (gelesen == buffer.length)
                ? buffer
                : Arrays.copyOf(buffer, gelesen);
        vearbeitePuffer(geleseneDaten);
    }
}
```

Listing 12.10 Textdaten lesen mit einem Puffer

So wird wesentlich effizienter gelesen als Zeichen für Zeichen. Mit jedem Aufruf von `read` wird der Puffer gefüllt. Der Rückgabewert ist die Anzahl Zeichen, die vom Stream gelesen wurden. Meist entspricht er der Größe des Puffers, es können aber weniger Zeichen gelesen werden, wenn das Ende der Daten erreicht ist oder gerade in diesem Moment nicht mehr Daten zur Verfügung stehen. Ist das Ende des Datenstroms er-

reicht, gibt `read` `-1` zurück; Daten werden in einer Schleife gelesen und verarbeitet, bis dieser Punkt erreicht ist.

Vorsicht ist geboten, wenn weniger Zeichen als die Puffergröße gelesen wurden. In diesem Fall wird nämlich der Rest des Puffers nicht verändert, am Ende des `char`-Arrays können Daten aus dem vorherigen Schleifendurchlauf stehen. Deswegen werden die gelesenen Daten, falls es weniger als die Puffergröße waren, in ein neues Array kopiert; so kann die Methode `verarbeitePuffer` immer mit einem vollständigen Array arbeiten und muss sich keine Sorgen um übrig gebliebene Daten am Ende des Arrays machen. Es ist zwar performanter, der verarbeitenden Methode das teilweise gefüllte Array und den Endindex zu übergeben, die gezeigte Variante ist aber weniger fehleranfällig, weil Sie in `verarbeitePuffer` nicht darauf achten müssen, wann Sie zu lesen aufhören.

Es ist sehr wichtig, dass Sie eine Datei nach dem Zugriff darauf wieder schließen. Im Beispiel geschieht das implizit durch das Statement try-with-resources, das auf seinen Ressourcen automatisch `close` aufruft. Sollten Sie aus irgendeinem Grund dieses Statement nicht verwenden können oder wollen, zum Beispiel weil Sie für eine ältere Java-Version entwickeln, dann müssen Sie selbst sicherstellen, dass der `Reader` (oder `Writer`, `InputStream`, `OutputStream` oder jedes Objekt, das auf eine Datei zugreift) ordentlich geschlossen wird.

```
public void liesAusDatei(File quelle) throws IOException{
    Reader reader = null;
    try {
        reader =  new BufferedReader(new FileReader(quelle));
        //Daten lesen und verarbeiten
    } finally {
        if (reader != null){
            reader.close();
        }
    }
}
```

Listing 12.11 Daten lesen mit traditionellem »try«-»finally«

Dieser Code ist etwas unhandlicher und hat zwei nicht zu vermeidende Unschönheiten. Die `Reader`-Variable muss außerhalb des `try`-Blocks deklariert werden, da `try` und `finally` keinen gemeinsamen Scope haben. Außerdem müssen Sie im `finally`-Block prüfen, ob der Reader nicht `null` ist. Das kann passieren, wenn schon beim Erzeugen des `Readers` eine Exception auftritt, beispielsweise eine `FileNotFoundException`, wenn die Datei nicht existiert. In diesem Fall gibt es keinen `Reader`, der geschlossen werden

kann, und ohne die entsprechende Prüfung käme es zu einer weiteren `NullPointerException`.

Ein weiteres Problem ist, dass auch `reader.close` eine `IOException` werfen kann. Im Beispiel werden innerhalb der Methode `liesAusDatei` keine Fehler behandelt, alle Fehler werden an den Aufrufer geworfen. Im schlimmsten Fall kann es so passieren, dass sowohl im `try`- als auch im `finally`-Block Fehler geworfen werden. Der Aufrufer erhält dann nur die Exception aus dem `finally`-Block, obwohl sie wahrscheinlich nur eine Konsequenz der Exception aus dem `try`-Block war.

Sie sehen also, der Code ist so länger, komplexer und fehleranfälliger. Es gibt keinen Grund, diese Variante zu bevorzugen, wenn Ihre Java-Version try-with-resources unterstützt.

Puffern und zeilenweise lesen

Das Puffern der Daten in einem `char[]` können Sie sich theoretisch sparen, wenn Sie einen `BufferedReader` einsetzen. Dessen Hauptaufgabe ist es nämlich, zu verhindern, dass Daten Byte für Byte von der Festplatte gelesen werden. Dazu liest er immer einen Puffer voll Daten ein, genau wie Sie es im Beispiel oben manuell machen, auch wenn Sie gerade nur ein Zeichen auslesen möchten. Nachfolgende `read`-Aufrufe werden dann aus dem Puffer bedient, solange dieser noch genügend Daten enthält, erst dann wird wieder auf die Festplatte zugegriffen.

Als Nebeneffekt seines Puffers hat der `BufferedReader` aber eine weitere sehr nützliche Fähigkeit: Er kann Textdaten zeilenweise lesen. `BufferedReader` bietet sowohl die Methode `readLine`, die die nächste Zeile der Datei liefert, als auch die Ihnen schon bekannte Methode `lines`, die alle Zeilen der Datei in einem Stream liefert. Wenn der Inhalt der zu lesenden Datei zeilenorientiert ist, wie zum Beispiel die Wetterdaten aus dem vorigen Kapitel, dann ist das natürlich viel praktischer, als Daten Zeichen für Zeichen oder Puffer für Puffer einzulesen und selbst nach den Umbrüchen zu suchen.

Sie können einen `BufferedReader` aus jedem anderen `Reader` erzeugen, indem Sie diesen als Parameter an den Konstruktor von `BufferedReader` übergeben:

```
try (BufferedReader reader = new BufferedReader(new FileReader(quelle))) {
    String zeile;
    while ((zeile = reader.readLine()) != null){
        verarbeiteZeile(zeile);
    }
}
```

Listing 12.12 Zeilenweise lesen

Der zugrunde liegende `FileReader` wird in einem `BufferedReader` verpackt, um die Fähigkeit zu puffern und zeilenweise zu lesen hinzuzufügen. Das ist eine Anwendung des *Decorator-Entwurfsmusters* (siehe Kasten), das für Ein- und Ausgabe in Java extensiv zum Einsatz kommt. Sie müssen in diesem Fall nur den `BufferedReader` schließen, dessen `close`-Methode ruft automatisch die `close`-Methode des dekorierten Readers auf.

Entwurfsmuster und das Decorator-Pattern

Entwurfsmuster sind Vorlagen, die zeigen, wie bestimmte Probleme in der Programmierung einfach und effektiv gelöst werden können. Sie sind ein fortgeschrittenes Thema der Softwareentwicklung, und nachdem Sie eine Programmiersprache beherrschen, ist es ein sinnvoller nächster Schritt, sich mit den wichtigsten Entwurfsmustern (engl. *design patterns*) zu beschäftigen. Eines dieser Pattern ist das *Decorator-Pattern*, das bei `Reader`, `Writer`, `InputStream` und `OutputStream` zum Einsatz kommt.

Das Decorator-Pattern dient dazu, Funktionalität zu Objekten hinzuzufügen, ohne auf Vererbung zurückzugreifen. Das klingt widersinnig, schließlich ist Vererbung doch ein Eckpfeiler der objektorientierten Programmierung, warum soll sie nun plötzlich vermieden werden?

Kein Werkzeug, egal, wie gut es ist, ist für jede Aufgabe geeignet, und im Fall der Ein-/Ausgabeklassen wäre Vererbung nicht die beste Wahl des Werkzeugs, weil zu viele spezialisierte Klassen entstünden. Das Problem ist gut an der Klasse `InputStream` zu sehen, die Binärdaten liest. (Dieselbe Argumentation trifft auch für `Reader` zu, aber `InputStream` illustriert den Punkt besser, weil es mehr Varianten der Klasse gibt.) Der grundlegende `InputStream` liest Daten aus verschiedenen Quellen. So gibt es einen `FileInputStream`, der aus einer Datei liest, einen `ByteArrayInputStream`, der Daten aus einem `byte[]` liest, und die Klasse `Socket` kann einen `InputStream` erzeugen, der aus einer Netzwerkverbindung liest. Genau wie `BufferedReader` gibt es einen `BufferedInputStream`, der einen Lesepuffer hinzufügt. Darüber hinaus gibt es einen `ObjectInputStream`, der Java-Objekte aus einem Datenstrom lesen kann (siehe Abschnitt 12.3, »Objekte lesen und schreiben«). Mit Vererbung müssten die grundlegenden Klassen alle mehrfach erweitert werden, um die möglichen Kombinationen herzustellen. Es gäbe dann: `FileInputStream`, `BufferedFileInputStream`, `ObjectFileInputStream`, `BufferedObjectFileInputStream`, `ByteArrayInputStream`, `BufferedByteArrayInputStream` usw. Mit dem Decorator-Pattern wird diese Flut von fast identischen Klassen vermieden, Sie können die gewünschte Kombination an Funktionen durch *Komposition* herstellen. Wenn Sie Java-Objekte gepuffert aus einer Datei lesen möchten, dann tun Sie das, indem Sie die Funktionalität »zusammendekorieren«: `new ObjectInputStream(new BufferedInputStream(new FileInputStream(datei)))`.

Schreiben in einen Writer

Das Schreiben in eine Datei funktioniert fast genauso wie das Lesen aus einer Datei. Sie erzeugen ein `FileWriter`-Objekt, dekorieren es vielleicht noch mit einem `BufferedWriter`, schreiben Ihre Daten hinein und schließen den `Writer` wieder.

```
try (BufferedWriter writer = new BufferedWriter(new FileWriter(ziel))) {
    for (String zeile : zeilen){
        writer.write(zeile);
        writer.newLine();
    }
}
```

Listing 12.13 Zeilenweise schreiben

So schreiben Sie Daten Zeile für Zeile in eine Textdatei. `Writer` sind in vielerlei Hinsicht das Spiegelbild von Readern. Sie haben eine Methode, die einzelne `char`-Werte schreibt, und eine Methode, die ein ganzes `char[]`schreibt – Sie können einen Schreibpuffer erzeugen, indem Sie Ihren `Writer` mit einem `BufferedWriter` dekorieren, und Sie müssen auch einen `Writer` in jedem Fall schließen, wenn Sie damit fertig sind. `Writer` selbst kennt auch wieder nicht das Konzept der Zeile. Wenn Sie zeilenweise schreiben möchten, dann ist der beste Weg, einen `BufferedWriter` und seine Methode `newLine` zu verwenden, um am Ende jeder Zeile einen Umbruch zu erzeugen.

Testen von I/O-Operationen

Ein Problem bei allen Arten von I/O-Operationen ist, dass sie nur schlecht testbar sind. Ihre Testfälle müssen sich darauf verlassen, dass bestimmte Dateien vorhanden sind und einen bestimmten Inhalt haben. Sie können ihnen zwar die entsprechenden Dateien beilegen, aber dann müssen Sie Dateien mit Ihren Testfällen ausliefern. Manchmal ist das nicht zu vermeiden, aber schön ist es nicht. Zum Glück gibt es einige Strategien, die das häufig vermeiden können.

Zunächst sollten Sie, wenn Sie Lese- und Schreiboperationen implementieren, niemals `File` als Methodenparameter deklarieren, sondern immer einen `Reader` oder `Writer` (bzw. `InputStream` oder `OutputStream`). Dadurch wird Ihr Code sofort testbarer, denn Sie können aus dem Testfall einen `StringReader` (oder `StringWriter`) übergeben. Es handelt sich dabei um einen vollwertigen `Reader` (bzw. `Writer`), der genau wie jeder andere `Reader` verwendet werden kann, aber seine Daten nicht aus einer Datei liest oder einer Netzwerkverbindung, sondern aus einem String, den Sie im Konstruktor übergeben. So hat Ihr Testfall die volle Kontrolle darüber, welche Daten die zu testende Methode zu sehen bekommt.

```
public static final String TESTDATEN =
 "2013\t0.2\t-0.7\t0.1\t8.1\t11.8\t15.7\t19.5\t17.9\t13.3\t10.6\t4.6\t3.6\n" +
 "2012\t1.9\t-2.5\t6.9\t8.1\t14.2\t15.5\t17.4\t18.4\t13.6\t8.7\t5.2\t1.5";

@Test
public void testLiesTemperaturdaten() {
    Reader testdaten = new StringReader(TESTDATEN);
    Temperaturstatistik statistik = Temperaturstatistik.liesDaten(testdaten);
    assertNotNull(statistik.getJahr(2013);
    //weitere Asserts folgen
}
```

Listing 12.14 Temperaturstatistik richtig testen

Der Testfall hängt so von keiner externen Datei ab; Sie definieren im Code, mit welchen Daten getestet werden soll. Sie können so auch für jeden Testfall andere Daten angeben, ohne mehrere Dateien anlegen zu müssen. Analog dazu funktioniert es auch mit dem Schreiben in einen `StringWriter`.

```
@Test
public void testSchreibePlayliste() {
    StringWriter testWriter = new StringWriter();
    Playlist playlist = new Playlist();
    playlist.addSong(...);
    playlist.schreibeNach(testWriter);
    assertEquals("erwarteter Inhalt", testWriter.toString);
}
```

Listing 12.15 Schreiboperationen testen

Die `toString`-Methode des `StringWriter` gibt alle Daten, die hineingeschrieben wurden, als einen String zurück. Sie können so in einem Testfall ganz leicht vergleichen, ob der Inhalt dem erwarteten Inhalt entspricht.

Analog zu diesen beiden Klassen gibt es `ByteArrayInputStream` und `ByteArrayOutputStream`, die dieselbe Aufgabe auf Basis eines `byte`-Arrays für Binärdaten erfüllen.

Etwas schwieriger wird es, wenn Ihr Testfall wirklich Dateien braucht, zum Beispiel weil die zu testende Methode Dateien in einem Verzeichnis suchen soll. Auch dafür ist es aber in vielen Fällen möglich, ohne mitgelieferte Dateien auszukommen, indem Sie selbst temporäre Dateien erzeugen. Die Methode `File.createTempFile` erzeugt eine Datei im Verzeichnis für temporäre Dateien Ihres Betriebssystems. Sie übergeben ein Präfix, das klarmachen sollte, woher diese Datei stammt, und eine Dateiendung. Als Rückgabewert erhalten Sie das `File`-Objekt der so angelegten Datei. Eine temporäre Datei wird aber nicht automatisch wieder gelöscht; um am Ende des

Tests wieder aufzuräumen, sollten Sie an jeder so erzeugten Datei noch `deleteOnExit` rufen, dann stellt Java sicher, dass diese Dateien auch wieder entfernt werden.

```
@Test
public void testMitDatei() throws IOException {
    File tempDatei = File.createTempFile(getClass().getName(), ".mp3");
    tempDatei.deleteOnExit();
    //Test durchführen
}
```

Listing 12.16 Eine temporäre Datei erzeugen

Gerade in Testfällen ist es eine gute Angewohnheit, den Klassennamen als Präfix für temporäre Dateien zu verwenden. So sieht man jeder Datei sofort an, woher sie stammt, falls sie zum Beispiel einmal nicht gelöscht wurde. Als optionalen dritten Parameter können Sie `createTempFile` ein Verzeichnis übergeben, in dem die Datei angelegt werden soll, falls Sie die Datei nicht im temporären Verzeichnis des Betriebssystems anlegen möchten.

Möchten Sie ein temporäres Verzeichnis anlegen, gibt es leider einmal mehr den hässlichen Bruch zwischen herkömmlichem `java.io` und `java.nio`: Die Methode, die temporäre Verzeichnisse anlegt, finden Sie nur in der `Files`-Klasse, dementsprechend erhalten Sie auch ein `Path`-Objekt zurück, aus dem Sie selbst wieder ein `File` machen müssen.

Weder für Verzeichnisse noch für Dateien müssen Sie sich übrigens Sorgen um die Eindeutigkeit Ihrer Dateinamen machen. In beiden Fällen ist dafür gesorgt, dass jeder Aufruf einen neuen Dateinamen erzeugt. Sie können also beliebig viele temporäre Dateien erzeugen, ohne dass es zu Konflikten kommt.

```
@Test
public void testMitVerzeichnis() throws IOException {
    File tempVerzeichnis = Files.createTempDirectory("mp3test").toFile();
    tempVerzeichnis.deleteOnExit();
    File tempDatei = File.createTempFile(getClass().getName(), ".mp3",
        tempVerzeichnis);
    tempDatei.deleteOnExit();
    //Test durchführen
}
```

Listing 12.17 Temporäre Verzeichnisse erzeugen

Alle gezeigten Klassen und Methoden haben natürlich auch Anwendungen außerhalb von Testfällen, aber Testfälle für I/O sind ohne sie nur sehr umständlich umzusetzen.

12.2.2 Übung: Playlisten – jetzt richtig

Einmal mehr sollen Sie eine `Playlist`-Klasse schreiben. Sie hat nichts mit den Klassen aus den vorigen Kapiteln zu tun, aber die weiteren Übungen in diesem Kapitel werden Sie in diese Richtung entwickeln. Für den Moment soll es in der Playliste eine Liste von Strings geben, die die absoluten Pfade zu Musikdateien enthält. Eine Methode `addSong` soll einen neuen Pfad hinzufügen, mit `getSongs` soll man die ganze Liste zurückbekommen.

Als Nächstes soll es durch eine statische Methode `ausVerzeichnis` möglich sein, eine neue Playliste zu erzeugen, die alle MP3-Dateien aus einem Verzeichnis und seinen Unterverzeichnissen enthält. Sie können dazu die Klasse `Musikfinder` aus der vorigen Übung weiterverwenden.

Eine Methode `schreibe` soll den Inhalt der Playliste in eine Datei schreiben, einen Pfad pro Zeile. Eine statische Methode `lese` soll eine solche Datei einlesen und wieder ein `Playlist`-Objekt daraus erzeugen.

Zuletzt soll eine Methode `verifiziere` für jeden Eintrag der Playliste prüfen, ob die Datei noch existiert, und den Eintrag aus der Liste entfernen, falls dem nicht so ist.

Schreiben Sie dann zwei Programme `PlaylistSchreiber` und `PlaylistChecker`. `PlaylistSchreiber` erwartet ein Verzeichnis und einen Dateinamen als Aufrufparameter, erzeugt eine `Playlist` aus dem Verzeichnis und speichert sie in der benannten Datei. `PlaylistChecker` erwartet eine `Playlist`-Datei als Parameter, liest diese ein, verifiziert sie und schreibt die bereinigte `Playlist` wieder in die Datei. Die Lösung zu dieser Übung finden Sie im Anhang.

12.2.3 »InputStream« und »OutputStream« – Binärdaten

Genau so, wie Sie mit `Reader` und `Writer` mit Textdateien umgehen, können Sie mit `InputStream` und `OutputStream` mit Binärdateien umgehen. Der einzige Unterschied ist, dass `Input-` und `OutputStream`s nicht mit `char` und `char[]` arbeiten, sondern mit `byte` und `byte[]`:

```
try (InputStream in = new FileInputStream(ziel)) {
    int gelesen;
    byte[] buffer = new byte[1024];
    while ((gelesen = in.read(buffer)) > -1) {
        byte[] geleseneDaten = (gelesen == buffer.length)
                ? buffer
                : Arrays.copyOf(buffer, gelesen);
        vearbeitePuffer(geleseneDaten);
    }
}
```

Listing 12.18 Binärdaten lesen

Analog zum `BufferedReader` gibt es für Binärdaten den `BufferedInputStream`, der einen Puffer zur Verfügung stellt. Und auch beim `OutputStream` gibt es keine Überraschungen; die `write`-Methode erwartet ein `byte[]` als Parameter, ansonsten bleibt alles gleich wie beim `Writer`.

Etwas schwieriger ist es, für Binärdaten einen nützlichen, aber dennoch übersichtlichen Anwendungsfall zu finden. Für Textdateien ist das einfach: Einen Text aus einer Datei können Sie anzeigen, in eine Datei geschriebenen Text können Sie im Texteditor verifizieren. Mit Binärdaten ist es schwieriger, einen anschaulichen Fall zu finden. Natürlich sind Audio- und Videodaten binär, aber sie zu decodieren und anzuzeigen ist äußerst aufwendig. Auch Programme sind Binärdateien, aber sie sind in einem Programm nicht wirklich sinnvoll zu verarbeiten. Sie werden in Abschnitt 12.3, »Objekte lesen und schreiben«, lernen, wie Sie Java-Objekte in einen Binärdatenstrom schreiben und aus einem solchen wieder auslesen.

Und ironischerweise sind es manchmal Textdaten, die Sie aus einem Datenstrom auslesen möchten. Nicht für alle Datenquellen gibt es `Reader` und `Writer`, eine Netzwerkverbindung können Sie zum Beispiel nur als Stream öffnen. Wenn Sie aber Textdateien aus einer solchen Netzwerkverbindung lesen möchten oder in sie hineinschreiben, dann müssen Sie aus dem `InputStream` einen `Reader` oder aus dem `OutputStream` einen `Writer` machen. Dazu dienen die Klassen `InputStreamReader` und `OutputStreamWriter`. Sie dekorieren einen `Input-` oder `OutputStream`, stellen sich aber nach außen als `Reader` und `Writer` dar.

```
Socket netzwerkverbindung = new Socket(…);
try (BufferedReader reader = new BufferedReader(
        new InputStreamReader(netzwerkverbindung.getInputStream()))){
    reader.lines().forEach(…);
}
```

Listing 12.19 Textdaten aus einer Netzwerkverbindung lesen

Aber auch wenn Sie mit Textdateien arbeiten, gibt es manchmal einen Grund, nicht direkt mit einem Reader aus einer Datei zu lesen, sondern mit einem `InputStream` und einem `InputStreamReader`: Sie können ein Encoding angeben. An anderer Stelle im Buch habe ich Encodings bereits erwähnt – es sind Abbildungen, die ein oder mehrere Byte Binärdaten einem Zeichen Textdaten zuweisen. Wenn Sie mit `FileReader` und `FileWriter` arbeiten, dann verwenden diese immer das Default-Encoding

Ihres Systems. Wenn Sie aber einen `InputStreamReader` oder `OutputStreamWriter` verwenden, dann können Sie selbst ein zu verwendendes Encoding angeben und so auch Dateien verarbeiten, die in einem anderen Encoding vorliegen. Sie müssen allerdings wissen, *welches* Encoding diese Datei hat.

```
FileInputStream fis = new FileInputStream(quelle);
try (BufferedReader reader =
 new BufferedReader(new InputStreamReader(fis, "Shift_JIS"))){
    reader.lines().forEach(...);
}
```

Listing 12.20 Text in einem japanischen Encoding lesen

Beim Versuch, eine Datei im japanischen Encoding Shift JIS mit einem `FileReader` zu lesen, käme nur Müll heraus – es sei denn, Ihr Computer benutzt Shift JIS als Default-Encoding. Indem Sie am `InputStreamReader` ein Encoding angeben, können Sie eine solche Datei dennoch korrekt einlesen.

12.2.4 Übung: ID3-Tags

Musik aus Musikdateien abzuspielen, ist zwar eine komplexe Aufgabe, aber eine andere Art von Information lässt sich vergleichsweise leicht aus vielen Dateien extrahieren. MP3-Dateien enthalten häufig ein sogenanntes *ID3-Tag*, einen kurzen Block von Textdaten, der Titel, Interpret und einige weitere Informationen enthält. Von diesem Tag gibt es zwei Versionen, von denen Version 1 (ID3v1) immer noch verbreitet und einfacher zu verarbeiten ist. Wenn ein ID3v1-Tag vorhanden ist, dann steht es in den 128 Byte der Datei und ist aufgebaut, wie in Tabelle 12.2 beschrieben. Alle Felder liegen im Encoding ISO-8859-1 vor und müssen nicht unbedingt befüllt sein.

Feld	Länge (in Byte)	Beschreibung
Header	3	Die Zeichenfolge »TAG«. Steht an dieser Stelle etwas anderes, dann ist kein ID3v1-Tag vorhanden, und Sie können die Verarbeitung abbrechen.
Titel	30	der Songtitel
Interpret	30	der Interpret
Album	30	das Album, auf dem der Song erschienen ist
Jahr	4	Erscheinungsjahr

Tabelle 12.2 Aufbau des ID3v1-Tags

Feld	Länge (in Byte)	Beschreibung
Kommentar	28 oder 30	Ein Freitext-Kommentar. Dieses Feld kann entweder 28 oder 30 Byte lang sein. Wenn es 28 Zeichen lang ist, dann sind die zwei übrigen Byte durch die beiden folgenden Felder belegt. Ist der Kommentar 30 Byte lang, entfallen die beiden folgenden Felder. Den Unterschied zwischen den beiden Varianten erkennen Sie daran, ob im nächsten Feld ein 0-Byte steht.
0-Byte	1	Hat der Kommentar 30 Byte, gehört dieses Feld noch zum Kommentar. Hat der Kommentar nur 28 Byte, steht hier ein Byte mit dem Wert 0, und das nächste Byte enthält die Tracknummer.
Track	1	Die Tracknummer des Songs auf dem Album.
Genre	1	Zahlencode für das Genre, z. B. 1 = Classic Rock, 22 = Death Metal und 75 = Polka. Eine vollständige Liste finden Sie unter *http://en.wikipedia.org/wiki/ID3*.

Tabelle 12.2 Aufbau des ID3v1-Tags (Forts.)

Schreiben Sie eine neue Klasse `Song`, oder erweitern Sie die schon vorhandene Klasse, die alle Felder des ID3v1-Tags und den absoluten Pfad zur Datei enthält. Schreiben Sie in dieser Klasse eine statische Methode `ausMP3`, die einen neuen Song erzeugt, bei dem alle Felder aus dem Tag befüllt sind, sofern es vorhanden ist.

Ändern Sie dann Ihre Playliste aus der vorigen Übung so ab, dass sie eine Liste von `Song`-Objekten enthält statt einer Liste von Strings. Beim Schreiben und Lesen der Playliste sollen alle Felder des Songs berücksichtigt werden. Geben Sie nach wie vor pro Song eine Zeile aus, und trennen Sie die einzelnen Felder durch `|`-Zeichen.

Ein Hinweis für den Anfang: Auch wenn Sie Textdaten auslesen wollen, handelt es sich doch um eine Binärdatei. Verwenden Sie also einen `InputStream`, und lesen Sie Byte-Werte aus. Aus ihnen können Sie mit dem String-Konstruktor `String(byte[] daten, Charset encoding)` einen String erzeugen, bei dem das richtige Encoding verwendet wird:

```
new String(buffer, Charset.forName("ISO-8859-1"));
```

Um die richtige Stelle in der Datei zu finden, müssen Sie nicht die ganze Datei einlesen und dann nur die letzten 128 Byte verarbeiten. Mit der `skip`-Methode können Sie eine Anzahl an Bytes überspringen und so gleich an der richtigen Stelle zu lesen beginnen.

Dies ist Ihre bisher schwierigste Aufgabe. Viel Erfolg!

Die Lösung zu dieser Übung finden Sie im Anhang.

12.3 Objekte lesen und schreiben

In der Übung des letzten Abschnitts haben Sie ein eigenes Dateiformat entworfen, um Playlisten zu speichern. Ein textbasiertes Format wie das dort verwendete hat den Vorteil, dass es auch von anderen Programmen gelesen werden kann, unabhängig von der Sprache, in der die Playlisten geschrieben sind. Dafür ist es aber auch manuell zu implementieren, was für Sie ein wenig Arbeit bedeutet.

12.3.1 Serialisierung

Java bietet unter dem Namen *Serialisierung* auch einen eigenen Mechanismus, mit dem Sie ganze Java-Objektbäume in einen Datenstrom schreiben und daraus wieder lesen können. Solche serialisierten Objekte sind aus einem Programm in einer anderen Sprache nicht lesbar, und auch ein Java-Programm kann sie nur lesen, wenn es die Originalklassen verwendet, die zur Serialisierung benutzt wurden. Es gibt aber auf der anderen Seite einige Vorteile, die die Nachteile wieder aufwiegen:

- Serialisierung ist sehr einfach zu implementieren.
- Serialisierung vermeidet einige Probleme mit selbst geschriebenen Textformaten, die wir in der Übung gekonnt ignoriert haben. Was wäre zum Beispiel, wenn ein Titel das |-Zeichen enthielte, das als Trenner verwendet wird? Solche Probleme lassen sich natürlich umgehen, aber der Implementierungsaufwand steigt dadurch weiter an.
- Serialisierung schreibt nicht nur Objekte in den Datenstrom, sondern auch ihre Beziehungen zueinander. Bei der Serialisierung werden alle Felder eines Objekts in den Strom geschrieben. Wenn es sich dabei um weitere Objekte handelt, dann werden auch sie serialisiert. Bei der *Deserialisierung*, also der Rückumwandlung in Java-Objekte, wird sichergestellt, dass die Referenzen zwischen den Objekten wieder originalgetreu hergestellt werden: Zwei Felder, die vor der Serialisierung dasselbe Objekt referenzierten, werden auch nach der Deserialisierung dasselbe Objekt referenzieren und nicht etwa verschiedene, identische Objekte.

Dabei ist Serialisierung wirklich einfach zu implementieren. Die einzige Voraussetzung ist, dass alle zu serialisierenden Objekte das Interface `Serializable` implementieren. Dabei handelt es sich um ein sogenanntes *Marker-Interface*; es enthält keine Methoden, sondern markiert nur Klassen, die serialisiert werden dürfen. Ist diese Voraussetzung erfüllt, dann müssen Sie das Objekt nur in einen `ObjectOutputStream` schreiben:


```
try (ObjectOutputStream oos =
 new ObjectOutputStream(new FileOutputStream(ziel))){
    oos.writeObject(playlist);
}
…
try (ObjectInputStream ois =
 new ObjectInputStream(new FileInputStream(quelle)){
    Playlist playlist = (Playlist) ois.readObject();
}
```

Listing 12.21 Ein Objekt serialisieren und deserialisieren

Beim Lesen aus einem `ObjectInputStream` müssen Sie selbst wissen, welchen Typ Sie erwarten, und entsprechend casten. Hier helfen Ihnen keine Generics, um den richtigen Typ zu raten.

Beachten sollten Sie auch noch, dass wirklich alle Objekte, die serialisiert werden, auch `Serializable` sind. Das trifft auch auf Objektfelder in den Objekten zu, die Sie serialisieren. Da sie auch in den Datenstrom geschrieben werden, müssen auch sie serialisierbar sein, sonst kommt es zu einer `NotSerializableException`, und die Operation schlägt fehl.

Das Feld »serialVersionUID«

Beim Serialisieren von Klassen wird ein `long`-Feld `serialVersionUID` in den Datenstrom geschrieben. Durch dieses Feld wird beim Deserialisieren sichergestellt, dass die Klasse, mit der das Objekt deserialisiert wird, sich seit der Serialisierung nicht verändert hat. Hat sich die Klasse seitdem verändert, dann ist das serialisierte Objekt nicht mehr lesbar, und es kommt zu einer `InvalidClassException`. Die automatisch berechnete `serialVersionUID` ist aber sehr empfindlich und kann auch bei einer unveränderten Klasse anders sein, wenn Ihr Programm zum Beispiel auf einer anderen JVM ausgeführt wird, obwohl das Objekt selbst lesbar wäre.

Um dieses Problem zu umgehen, können Sie selbst eine `serialVersionUID` für Ihre Klasse angeben, indem Sie eine Konstante deklarieren:

```
private static final long serialVersionUID = 23L;
```

Oracle empfiehlt, dieses Feld in jeder Klasse anzugeben, um die genannten Probleme zu vermeiden. Sie handeln sich aber damit das neue Problem ein, dass es nun Ihre Aufgabe ist, die `serialVersionUID` zu ändern, wenn Sie Änderungen an der Klasse vornehmen und Felder hinzufügen oder entfernen.

Glücklicherweise haben Sie etwas Einfluss darauf, wie Ihre Objekte serialisiert werden. Soll ein Feld nicht serialisiert werden, weil es nicht `Serializable` ist, oder aus an-

deren Gründen, können Sie dieses Feld als `transient` deklarieren, und es wird nicht in den Datenstrom geschrieben:

```
class Playlist implements Serializable {
    private List<Song> songs = …;
    private transient Long gesamtlaenge = …;
}
```

Listing 12.22 Ein transientes Feld

Über den Sinn und Unsinn, einen `Long`-Wert nicht zu serialisieren, lässt sich streiten. Er wäre serialisierbar, er wird in diesem Kontext aber nicht serialisiert. Ein möglicher Grund könnte sein, dass der Wert, wenn die Playliste geladen wird, neu berechnet werden soll, um einen korrekten Wert zu finden, falls zwischenzeitlich Musikdateien gelöscht wurden.

Sie können sogar noch tieferen Einfluss auf den Serialisierungsprozess nehmen, indem Sie Methoden mit den folgenden Signaturen implementieren:

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException;
private void readObject(java.io.ObjectInputStream in)
 throws IOException, ClassNotFoundException;
```

Listing 12.23 Mehr Einfluss auf den Serialisierungsprozess

Diese Methoden haben einen Hauch von schwarzer Magie. Sie werden von keinem Interface gefordert, sie werden von nirgendwo überschrieben, aber wenn sie mit der richtigen Signatur existieren, dann werden sie bei der Serialisierung oder Deserialisierung aufgerufen und geben Ihnen die Möglichkeit, eigenen Code auszuführen. So können Sie zum Beispiel bei der Deserialisierung dafür sorgen, dass transiente Felder neu berechnet werden und sofort nach der Deserialisierung zur Verfügung stehen:

```
class Playlist implements Serializable {
    private List<Song> songs = …;
    private transient Long gesamtlaenge = …;
    private void readObject(java.io.ObjectInputStream in)
             throws IOException, ClassNotFoundException{
        in.defaultReadObject();
        verifiziere();
        berechneGesamtlaenge();
    }
}
```

Listing 12.24 Transiente Felder neu berechnen

Die Methode `defaultReadObject` des `ObjectInputStream` führt die »normale« Deserialisierung aus, anschließend wird eigener Code ausgeführt, nicht mehr existierende Dateien werden entfernt, und die Gesamtlänge wird neu berechnet. Damit befinden Sie sich allerdings schon tief im Bereich der fortgeschrittenen Themen.

Und was ist jetzt besser?

Was ist besser, ein eigenes Datenformat oder serialisierte Objekte? Das ist eine dieser Fragen, auf die es entweder keine Antwort gibt oder zu viele. Sie müssen für den konkreten Fall entscheiden, was Ihnen wichtiger ist: einfache und robuste Programmierung oder einfacher Datenaustausch mit anderen Programmen.

Oder vielleicht sogar eine dritte Möglichkeit: Für Playlisten gibt es mit M3U bereits ein Format, das von vielen Programmen verstanden wird. Würde Ihr Programm M3U-Playlisten erstellen, dann könnten Sie diese auch in iTunes oder dem Windows Media Player öffnen. Dafür ist der Implementierungsaufwand noch etwas höher, denn Sie müssen zunächst das korrekte Format für eine solche Datei ermitteln.

So ungern ich die Antwort gebe, aber welche Ausgabe besser ist, »hängt davon ab«.

12.4 Netzwerkkommunikation

Bisher ging es nur um Ein- und Ausgabe mit Dateien, aber wie bereits angedeutet, gibt es in Java keinen nennenswerten Unterschied zwischen I/O mit Dateien und I/O mit anderen Quellen wie Netzwerkverbindungen. In allen Fällen basiert die Ein- und Ausgabe auf `InputStream` und `OutputStream`, der Unterschied liegt nur darin, wo diese Datenströme herkommen. Bei der Netzwerkkommunikation mit dem TCP-Protokoll kommen sie aus einem `Socket`. (Kommunikation mit UDP verwendet die Klasse `DatagramSocket` und basiert nicht auf Streams.)

`Socket` hat zwar eine lange Liste von Methoden, aber für die grundlegende Verwendung können Sie die meisten davon ignorieren.

```
String nachricht = in.readLine();
try (Socket verbindung = new Socket("localhost", 23456)){
    BufferedReader reader = new BufferedReader(
     new InputStreamReader(verbindung.getInputStream()));
    BufferedWriter writer = new BufferedWriter(
     new OutputStreamWriter(verbindung.getOutputStream()));
    writer.write(nachricht);
    writer.newLine();
```

```
    writer.flush();
    String antwort = reader.readLine();
}
```

Listing 12.25 Netzwerkkommunikation mit einem Socket

Sie geben dem Socket im Konstruktor Adresse (IP-Adresse oder Hostname) und Port des Servers an, mit dem Sie sich verbinden möchten. Die Verbindung wird automatisch hergestellt, und mit den Methoden `getInputStream` und `getOutputStream` können Sie Daten vom Server empfangen und zum Server senden.

Einen kleinen Unterschied zwischen Netzwerkkommunikation und Datei-I/O gibt es mit der `flush`-Methode. Sie sorgt dafür, dass der Schreibpuffer sofort weiterverarbeitet wird, auch wenn er noch nicht voll ist. Bisher haben Sie diese Methode nicht benötigt, weil der Puffer auch geleert wird, wenn Sie den Datenstrom schließen. Hier wird der Strom aber nicht sofort geschlossen, denn es sollen nicht nur Daten in eine Richtung versendet werden, es soll echte Kommunikation in beide Richtungen stattfinden. Damit der Server eine Antwort schicken kann, die Sie dann mit `readLine` lesen können, muss er zunächst Ihre Nachricht erhalten, und dazu müssen Sie den Puffer leeren.

Ihnen wird außerdem nicht entgangen sein, dass weder `InputStream` noch `OutputStream` geschlossen werden. Beide sind fest mit dem Socket verbunden, aus dem sie hergestellt wurden, und wenn Sie einen der Ströme schließen, wird der Socket geschlossen. Andersherum werden die Datenströme aber auch geschlossen, wenn Sie den Socket schließen, deswegen reicht es, diesen als Ressource für den `try`-Block anzugeben.

So sieht es auf der Clientseite der Kommunikation aus, aber wie ist es mit der Serverseite? Ein einfaches Serverprogramm in Java zu schreiben, ist kaum anders als ein Clientprogramm, nur wo der Socket herkommt, ändert sich.

```
ServerSocket server = new ServerSocket(23456);
try (Socket verbindung = server.accept()){
    BufferedReader reader = new BufferedReader(
     new InputStreamReader(verbindung.getInputStream()));
    BufferedWriter writer = new BufferedWriter(
     new OutputStreamWriter(verbindung.getOutputStream()));
    String nachricht = reader.readLine();
    writer.write(antwort);
    writer.flush();
}
```

Listing 12.26 Netzwerkkommunikation von der Serverseite

Ein `ServerSocket` dient nicht direkt der Kommunikation, er wartet nur auf eingehende Verbindungen. Der Konstruktorparameter gibt den Port an, auf dem Verbindungen akzeptiert werden sollen; die Methode `accept` wartet, bis auf diesem Port eine Verbindung hergestellt wird. Und warten heißt hier wirklich warten: `accept` blockiert so lange, bis eine Verbindung aufgebaut wird. Wenn eine Verbindung zustande kommt, gibt `accept` einen `Socket` zurück, mit dem Sie genau so verfahren können wie mit einem `Socket` auf der Clientseite.

Wie gezeigt, wird nur eine Verbindung akzeptiert und verarbeitet. Für ein Beispiel ausreichend, für einen echten Serverprozess werden dagegen üblicherweise Verbindungen in einer Schleife akzeptiert, und die Verarbeitung wird in einem neuen Thread durchgeführt, so dass dieser Thread erneut mit `accept` auf Verbindungen warten kann.

```
ServerSocket server = new ServerSocket(23456);
while(!beendet){
    try (Socket verbindung = server.accept()){
        new Thread(() -> verarbeiteVerbidnung(verbindung));
    }
}
```

Listing 12.27 »ServerSocket« mit Threads

12.4.1 Übung: Dateitransfer

Für diese Übung müssen Sie zwei Programme schreiben, einen Server und einen Client. Der Server soll mit einem Port und einem Verzeichnis als Aufrufparameter gestartet werden. Es soll auf dem Port auf eine Verbindung warten, von dieser Verbindung einen Dateinamen und den Dateiinhalt lesen und diese Datei in das übergebene Verzeichnis schreiben.

Die Aufgabe des Clients ist damit schon klar: Er soll mit einem Hostnamen, Port und Dateinamen aufgerufen werden, eine Verbindung zu diesem Host auf diesem Port aufbauen und Dateinamen und Inhalt übermitteln.

Lesen und Schreiben der Dateien sind nichts Neues mehr, und mit den Netzwerkverbindungen können Sie, wie Sie gesehen haben, sehr ähnlich umgehen. Insofern müssen Sie nur das in diesem Kapitel Gelernte zusammensetzen.

Eine Schwierigkeit ist, dass Sie ein Protokoll entwickeln müssen, um in einem Stream zuerst den Dateinamen und dann den Dateiinhalt zu übertragen. Es ist kein umfangreiches oder komplexes Protokoll, aber es ist ein Protokoll, das beide Seiten benötigen, um zu kommunizieren. Die einfachste Lösung dafür: Zwischen Dateinamen und Inhalt wird ein einzelnes 0-Byte gesendet. In einem gültigen Dateinamen kann kein Byte mit dem Wert 0 vorkommen, es eignet sich deshalb gut als Terminator. Alles,

was nach dem ersten 0-Byte steht, ist der Inhalt der Datei. Als weitere Einschränkung sollte der Dateiname niemals länger als 255 Byte sein, das ist das Limit vieler aktuell verbreiteter Dateisysteme.

Idealerweise können Sie Ihre Programme mit zwei verschiedenen Computern testen, so dass die Daten wirklich über ein Netzwerk übertragen werden. Wenn Sie diese Möglichkeit nicht haben, können Sie aber auch beide Programme auf demselben Computer ausführen und als Hostnamen für den Client `localhost` angeben. Die Lösung zu dieser Übung finden Sie im Anhang.

12.5 Zusammenfassung

Sie haben in diesem Kapitel den letzten wichtigen Baustein kennengelernt, um »echte« Programme schreiben zu können: die Ein- und Ausgabe von Daten. Sie haben gelernt, wie Sie in Java mit Dateien und Verzeichnissen arbeiten und wie Sie Text- und Binärdaten lesen und schreiben. Sie haben gesehen, wie Sie mit Netzwerkverbindungen arbeiten und dass es keinen nennenswerten Unterschied zwischen Netzwerk-I/O und Datei-I/O gibt.

Um einen speziellen Teil der Netzwerkprogrammierung, die Servlet-Technologie für HTTP-Server in Java, wird es in Kapitel 14 gehen. Im nächsten Kapitel beschäftigen wir uns zunächst mit einem anderen fortgeschrittenen Thema der Programmierung: dem Multithreading, der Ausführung eines Programms in mehreren Ausführungssträngen.

Die beiden geforderten Statistiken berechnen Sie dann jeweils mit einem `Collector`:

```
private static void monatsdurchschnitt(String dateiname) throws
IOException {
    System.out.println("Temperaturen im Monatsdurchschnitt");
    try (BufferedReader reader = new BufferedReader(
     new FileReader(dateiname))){
        Map<Month, Double> statistik = erzeugeMonatswertStream(reader)
         .collect(Collectors.groupingBy(mw -> Month.of(mw.monat + 1),
          TreeMap::new,
          Collectors.averagingDouble(mw -> mw.temperatur)));
    for (Map.Entry<Month, Double> entry : statistik.entrySet())
        System.out.println(entry.getKey() + ": " + entry.getValue() + " Grad");
    }
}
```

Listing B.66 Berechnung der Monatsdurchschnitte

Sie suchen nach dem Durchschnitt pro Monat, also müssen Sie die Werte zunächst gruppieren. Der entsprechende `groupingBy`-Collector gruppiert nach Monatsobjekten, nicht nach den `int`-Werten, einfach um anschließend die Ausgabe des Monatsnamens zu vereinfachen. Er macht außerdem von der Möglichkeit Gebrauch, mittels eines Suppliers (`TreeMap::new`) eine Map-Implementierung vorzugeben. `TreeMap` ist eine `SortedMap`, für die anschließende Ausgabe muss deshalb nur über die `Map` iteriert werden, und die Monate werden in der richtigen Reihenfolge ausgegeben. Außerdem wird ein weiterer Collector übergeben, der auf die Gruppe angewendet wird und ihren Durchschnitt berechnet. So entsteht eine `Map<Month, Double>` mit zwölf Einträgen.

Um den Jahresdurchschnitt zu berechnen, muss lediglich eine andere Gruppierungsfunktion übergeben werden: **`mw -> mw.jahr`**. Das heißt, dass Sie sogar noch mehr Gemeinsamkeiten in eine Methode hätten auslagern können, aber die Methode `erzeugeMonatswertStream` ist wie gezeigt vielseitiger, mit einem Stream von `Monatswert`-Objekten können Sie mehr tun, als nur Statistiken zu generieren.

Lösung zu 12.1.3: Dateien kopieren

In dieser Aufgabe war ein wenig von allem drin: `File`-Objekte aus `String` erstellen, Existenz und Rechte prüfen, Verzeichnisse anlegen und `Files`-Methoden nutzen. Der grobe Ablauf des Programms war klar:


```
public static void main(String[] args) {
    if (args.length != 2){
        endeMitFehler("Sie müssen 2 Dateien angeben");
    }
    try {
        File quelle = new File(args[0]);
        File ziel = new File(args[1]);
        pruefeQuellDatei(quelle);
        ziel = pruefeUndErzeugeZiel(ziel, quelle.getName());
        File kopie = Files.copy(quelle.toPath(), ziel.toPath()).toFile();
    } catch (Exception e){
        endeMitFehler(e.getMessage());
    }
}
```

Listing B.67 Der Ablauf des Kopierprogramms

Zuerst wird selbstverständlich auch hier geprüft, ob die richtigen Aufrufparameter übergeben wurden. Falls dem nicht so ist, wird mit der Hilfsmethode `endeMitFehler` eine Fehlermeldung ausgegeben und das Programm beendet. Dieselbe Methode wird auch verwendet, um eine Ausgabe zu machen, falls beim Kopieren der Datei Exceptions auftreten. In diesem Fall ist es auch in Ordnung, allgemeine Exceptions zu fangen und nicht nur bestimmte Spezialisierungen, schließlich soll für jeden Fehler eine Ausgabe gemacht werden.

Stimmt die Anzahl der Parameter, werden aus Quelle und Ziel `Files` erzeugt. Für beide wird geprüft, ob die Angaben gültig sind – Code folgt sofort –, und dann die Kopie ausgeführt. Der Rückgabewert von `Files.copy` wird ignoriert, denn wäre die Kopie fehlgeschlagen, dann hätte die Methode eine Exception geworfen, es muss also nicht geprüft werden, ob die neue Datei wirklich existiert.

Bei der Prüfung der Quelldatei ist nicht viel zu tun, Sie müssen lediglich sicherstellen, dass die Datei gelesen werden kann:

```
private static void pruefeQuellDatei(File quelle) throws
Exception {
    if (!quelle.exists()){
        throw new Exception("Quelle nicht vorhanden");
    }
    if (!quelle.isFile()){
        throw new Exception("Quelle ist keine Datei");
    }
```

```
    if (!quelle.canRead()){
        throw new Exception("Quelle nicht lesbar");
    }
}
```

Listing B.68 Die Quelldatei prüfen

Die Datei muss existieren, es muss sich um eine Datei handeln, nicht um ein Verzeichnis, und sie muss lesbar sein. Wenn diese Voraussetzungen erfüllt sind, dann kann die Kopie von diesem Ende aus losgehen. Etwas mehr zu tun gibt es auf der Zielseite:

```
private static File pruefeUndErzeugeZiel(File ziel, String name) throws
Exception {
    if(ziel.exists() && ziel.isFile()){
        throw new Exception("Zieldatei existiert bereits");
    } else if (ziel.exists() && ziel.isDirectory()){
        if (!ziel.canWrite()){
            throw new Exception("Zielverzeichnis nicht schreibbar");
        } else {
            return new File(ziel, name);
        }
    } else {
        //Ziel existiert nicht
        ziel.getParentFile().mkdirs();
        return ziel;
    }
}
```

Listing B.69 Die Zieldatei vorbereiten

Für die Zieldatei ist zu prüfen, welcher der beschriebenen Fälle zutrifft. Existiert das Ziel bereits und ist es eine Datei, wird ein Fehler geworfen. Existiert es als Verzeichnis und ist schreibbar, dann ist das wahre Ziel der Kopie eine Datei in diesem Verzeichnis, die den Namen der Quelldatei hat. Dieses Ziel ist mit `new File(ziel, name)` leicht zu erzeugen.

Für den Fall, dass die Zieldatei noch nicht existiert, müssen Sie sicherstellen, dass das Elternverzeichnis existiert. Hier zahlt es sich aus, wenn Sie ein wenig im Javadoc gestöbert haben, denn die Methode `mkdirs` nimmt Ihnen diese Arbeit ab: Sie legt alle Verzeichnisse eines Pfades an, die noch nicht existieren. So wird das übergeordnete Verzeichnis des Ziels angelegt, denn das letzte Pfadelement soll in diesem Fall der neue Dateiname sein, nicht ein weiteres Verzeichnis.

Lösung zu 12.1.5: Musik finden

Dass Sie `Files.find` nicht benutzen können, klingt zunächst tragisch, aber die Rekursion durch sämtliche Unterverzeichnisse ist zum Glück nicht sehr aufwendig selbst zu implementieren:

```
public class Musikfinder {
    final private File start;
    public Musikfinder(String start){
        this(new File(start));
    }
    public Musikfinder(File start){
        if (!start.exists() || !start.isDirectory() || !start.canRead()){
            throw new IllegalArgumentException("Startverzeichnis muss ein
            lesbares Verzeichnis sein");
        }
        this.start = start;
    }

    public void findeMusik(Consumer c){
        findeMusik(start, c);
    }

    private void findeMusik(File verzeichnis, Consumer c) {
        File[] unterverzeichnisse = verzeichnis.listFiles(f ->
         f.isDirectory() && f.canRead());
        if (unterverzeichnisse != null){
            Arrays.stream(unterverzeichnisse).forEach(f ->
             this.findeMusik(f, c));
        }
        File[] musikDateien = verzeichnis.listFiles(f->
         f.getName().endsWith(".mp3"));
        if (musikDateien != null){
            Arrays.stream(musikDateien).forEach(c);
        }
    }
}
```

Listing B.70 Die Klasse »Musikfinder«

Ein `Musikfinder`-Objekt bekommt im Konstruktor entweder einen String oder ein `File`-Objekt, das ist ein sehr häufiger Fall von *Constructor Chaining*, wenn Sie mit Dateien arbeiten. Der Konstruktor prüft, ob ein lesbares Verzeichnis übergeben wurde,

sonst tut er nichts. Zum Start der Suche wird die Methode `findeMusik(Consumer)` gerufen, aber sie delegiert nur an `findeMusik(File, Consumer)` mit dem Startverzeichnis.

In dieser Methode passiert die Arbeit. Zunächst findet sie vom übergebenen Verzeichnis alle lesbaren Unterverzeichnisse und ruft für jedes rekursiv wieder `findeMusik`. Danach sucht sie alle Dateien mit der Endung *.mp3* und ruft für jede Datei den übergebenen `Consumer` auf.

Die `null`-Prüfung für das Ergebnis von `listFiles` ist leider notwendig, denn die Methode kann `null` zurückgeben, wenn bei der Ausführung eine `IOException` auftrat. Das kann zum Beispiel wieder für fehlende Rechte passieren und muss deshalb abgefangen werden.

Lösung zu 12.2.2: Playlisten – jetzt richtig

Die gesamte Klasse abzudrucken, ist inzwischen wohl nicht mehr notwendig; die Methoden, die Songs hinzufügen und die Liste auslesen, enthalten nichts Neues. Konzentrieren wir uns lieber auf die neuen Themen, zunächst das Schreiben:

```
public void schreibe(File ziel) throws IOException {
    schreibe(new FileWriter(ziel));
}

public void schreibe(Writer ziel) throws IOException {
    try (BufferedWriter buffered = new BufferedWriter(ziel)) {
        for (String song : songs) {
            buffered.write(song.toString());
            buffered.newLine();
        }
    }
}
```

Listing B.71 Playlistendateien schreiben

Dass es zwei `schreibe`-Methoden gibt, ist ein häufiges Muster: Die `schreibe`-Methode mit einem `Writer`-Parameter ist gut testbar, die Methode mit `File`-Parameter ist praktischer für den Gebrauch aus einem Programm. Der Schreibvorgang ist sehr einfach, es wird für jeden Pfad aus der Liste eine Zeile ausgegeben. Vergessen Sie nicht, den `Writer` zu schließen.

Als Nächstes sollen so erzeugte Dateien wieder eingelesen werden:

```
public static Playlist lese(File quelle) throws IOException{
    return lese(new FileReader(quelle));
}
```



```
public static Playlist lese(Reader quelle) throws IOException{
    try (BufferedReader reader = new BufferedReader(quelle)){
        Playlist playlist = new Playlist();
        reader.lines().forEach(playlist::addSong);
        return playlist;
    }
}
```

Listing B.72 Playlistendateien lesen

Auch hier gibt es wieder zwei Methoden, eine mit `File`-Parameter und eine mit `Reader`. Auch diese Methoden sind kurz und einfach, sie erzeugen eine neue Playliste und fügen Zeile für Zeile Einträge hinzu. Für das Erzeugen der Playliste aus einem Verzeichnis war ein großer Teil der Arbeit schon getan, der Musikfinder hat alles, was Sie dafür brauchen:

```
public static Playlist ausVerzeichnis(File startVerzeichnis) {
    Playlist playlist = new Playlist();
    new Musikfinder(startVerzeichnis).findeMusik(datei ->
     playlist.addSong(datei.getAbsolutePath()));
    return playlist;
}
```

Listing B.73 Playliste aus einem Verzeichnis erstellen

Da der Musikfinder vorausschauend so entwickelt war, dass Sie mit einem `Consumer` übergeben können, was mit gefundenen Dateien passiert, ist nur ein solcher `Consumer` zu übergeben, der der Playliste Einträge hinzufügt. Fehlt als Letztes noch die Methode zur Verifikation:

```
public int verifiziere(){
    int vorher = songs.size();
    songs = songs.stream()
            .filter(song -> new File(song).exists())
            .collect(Collectors.toList());
    return vorher - songs.size();
}
```

Listing B.74 Die Playliste ausmisten

Hier gab es sehr viele Methoden, die Anforderung umzusetzen. Wie gezeigt werden alle nicht mehr gefundenen Dateien aus der Liste herausgefiltert, und das Filterergebnis wird als neue Liste gesetzt.

Für die beiden Programme, die Playlisten erzeugen und überprüfen, waren nur noch Aufrufparameter zu prüfen und `Playlist`-Methoden aufzurufen. Sie sehen den Quellcode der Programme in den Downloads (*www.rheinwerk-verlag.de/4096*). Schauen Sie sich dort auch die Testfälle an, die die beschriebenen Techniken zum Testen von I/O-Operationen demonstrieren.

Lösung zu 12.2.4: ID3-Tags

Diese Übung war recht anspruchsvoll, umso überraschender ist es, wie kurz die Lösung schlussendlich ist. Wie die `Song`-Klasse mit all ihren Feldern aussieht, ist nichts Neues mehr, interessant ist der Code, der die Felder befüllt:

```
private static final Charset ISO_8859_1 = Charset.forName("ISO-8859-1");
private static final byte[] TAG = "TAG".getBytes(ISO_8859_1);
public static Song ausMP3(File mp3) throws IOException {
    if (mp3 == null || !mp3.exists() || !mp3.isFile() || !mp3.canRead())
        throw new IllegalArgumentException("mp3 muss eine lesbare Datei sein.");
    Song song = new Song();
    song.pfad = mp3.getAbsolutePath();
    try (InputStream in = new BufferedInputStream(new FileInputStream(mp3))){
        in.skip(mp3.length() - 128);
        byte[] buffer = new byte[3];
        in.read(buffer);
        if (Arrays.equals(buffer, TAG)){
            fuelleSongAusStream(song, in);
        }
    }
    return song;
}

private static void fuelleSongAusStream(Song song, InputStream in) throws
IOException {
    byte[] buffer = new byte[30];
    in.read(buffer);
    song.setTitel(new String(buffer, ISO_8859_1).trim());
    in.read(buffer);
    song.setInterpret(new String(buffer, ISO_8859_1).trim());
    in.read(buffer);
    song.setAlbum(new String(buffer, ISO_8859_1).trim());
    in.read(buffer, 0, 4);
```



```
    String jahrAlsString = new String(buffer, 0, 4, ISO_8859_1);
    if (jahrAlsString.matches("\\d{4}")){
        song.setJahr(Integer.parseInt(jahrAlsString));
    }
    in.read(buffer);
    if (buffer[28] == 0){
        //Track-Nummer vorhanden
        song.setKommentar(new String(buffer, 0, 28, ISO_8859_1).trim());
        song.setTrack(Integer.valueOf(buffer[29]));
    } else {
        //Track-Nummer nicht vorhanden
        song.setKommentar(new String(buffer, ISO_8859_1).trim());
        song.setTrack(null);
    }
    song.setGenre((byte) in.read());
}
```

Listing B.75 Informationen aus dem ID3-Tag auslesen

In der Methode `ausMP3` selbst passiert noch nicht viel, es wird der Pfad der Datei gesetzt und geprüft, ob überhaupt Tag-Daten vorhanden sind. Dazu wird der Dateiinhalt übersprungen bis zur Stelle 128 Byte vor dem Ende, dann werden 3 Byte gelesen und diese mit einer Konstanten verglichen, die das `byte[]` zum String `"TAG"` enthält. Steht an dieser Stelle wirklich die Zeichenfolge TAG, so werden in `fuelleSongAusStream` die Felder des Objekts befüllt, ansonsten passiert nichts, und die Felder bleiben leer.

In `fuelleSongAusStream` werden die einzelnen Feldinhalte in ein `byte[]` gelesen und mit dem vorgegebenen Encoding ISO-8859-1 in einen String umgewandelt. Alle so erzeugten Strings werden noch mit `trim` gekürzt, da sie sonst in einer Reihe von Leerzeichen enden.

Zusätzliche Prüfungen sind an zwei Stellen nötig. Das Jahr kann nur als Zahl geparst werden, wenn es auch im Tag vorhanden war. Stand dort nur ein Leer-String, würde `Integer.parseInt` fehlschlagen, deshalb wird vorher mit einem regulären Ausdruck geprüft, ob wirklich eine vierstellige Zahl gelesen wurde.

Das Kommentarfeld wird immer mit 30 Zeichen eingelesen. Erst dann wird geprüft, ob an vorletzter Stelle der Wert 0 steht, und entsprechend reagiert.

Die Änderungen an `Playlist`, um die zusätzlichen Felder zu schreiben und zu lesen, sind danach mehr Fleißarbeit und enthalten nichts Neues mehr. Schauen Sie bei Fragen in den Beispielcode.

Lösung zu 12.4.1: Dateitransfer

Neben dem Aufbau der Netzwerkverbindung gab es in dieser Übung nichts wirklich Neues, aber alles Gelernte aus dem Kapitel musste zu einem recht komplexen Programm zusammengeführt werden. Deshalb wollen wir beide Programme, Client und Server, etwas detaillierter betrachten. Ausgespart wird dabei die Verarbeitung der Aufrufparameter, die Sie inzwischen zur Genüge kennen. Beginnen wir auf der etwas einfacheren Clientseite:

```
public void sendeDatei(File datei) throws IOException {
    try (Socket verbindung = new Socket(hostname, port)) {
        BufferedOutputStream out =
         new BufferedOutputStream(verbindung.getOutputStream());
        sendeDateinamen(out, datei);
        sendeInhalt(out, datei);
    }
}
```

Listing B.76 Datei versenden

Hostname und Port sind als Felder des Clientobjekts gesetzt, nur die Datei wird als Parameter der `sendeDatei`-Methode übergeben. Sie könnten so denselben Client benutzen, um mehrere Dateien zu versenden. Es wird zuerst ein Socket zum Server geöffnet. Dabei müssen Sie sich keine Gedanken darüber machen, ob als Hostname eine IP-Adresse, ein DNS-Name oder ein fester Name wie »localhost« übergeben wurde, die `Socket`-Klasse übernimmt diese Arbeit für Sie. Es wird ein `BufferedOutputStream` in den Socket erzeugt und dann zunächst der Dateiname und anschließend der Dateiinhalt in den Strom geschrieben. Im Detail sieht das so aus:

```
private void sendeDateinamen(BufferedOutputStream out, File datei) throws
IOException {
    String name = datei.getName();
    if (name.length() > 255) {
        name = name.substring(0, 255);
    }
    out.write(name.getBytes(ISO_8859_1));
    out.write(0);
}
```

Listing B.77 Den Dateinamen senden

Hier passiert nichts Besonderes; der Dateiname wird, falls notwendig, auf 255 Zeichen gekürzt und dann als `byte[]` in den Stream geschrieben. Der Name wird im Encoding ISO-8859-1 übertragen, weil dort praktischerweise jedes Zeichen einem Byte entspricht und die Längenbegrenzung in Zeichen und Byte somit gleich ist. Ein Nachteil davon ist, dass Sie nur Dateinamen in mitteleuropäischen Schriftsystemen übermitteln können, was aber für unsere Zwecke reichen sollte. Sie könnten auch ein anderes Encoding, wie UTF-16, verwenden. Dann würde die Längenprüfung zwar ein wenig komplizierter, aber wichtig ist am Ende nur, dass beide Programme dasselbe Encoding benutzen. In diesem Fall ist es aber wichtig, dass Sie ein Encoding angeben und sich nicht auf das Default-Encoding verlassen. Die Programme könnten auf unterschiedlichen Systemen laufen, die unterschiedliche Encodings verwenden. Um den Dateiinhalt zu senden, müssen Sie anschließend nur Daten aus der Datei in den Socket schreiben:


```
private void sendeInhalt(BufferedOutputStream out, File datei) throws
IOException {
    try (InputStream in = new BufferedInputStream(
     new FileInputStream(datei))) {
        byte[] buffer = new byte[4096];
        int gelesen = 0;
        while ((gelesen = in.read(buffer)) != -1) {
            out.write(buffer, 0, gelesen);
        }
    }
}
```

Listing B.78 Den Dateiinhalt senden

Es werden jeweils 4 kB aus der Datei gelesen und in den Socket geschrieben, bis als Zahl gelesener Bytes –1 zurückkommt, das Signal, dass der Datenstrom am Ende ist. Das Lesen auf der Serverseite ist etwas schwieriger, aber auch nicht problematisch:

```
public void erwarteVerbindung() throws IOException {
    ServerSocket server = new ServerSocket(port);
    try (Socket verbindung = server.accept()) {
        BufferedInputStream in =
         new BufferedInputStream(verbindung.getInputStream());
        String dateiname = liesNameAusStream(in);
        File datei = erzeugeSichereDatei(dateiname, verzeichnis);
        schreibeDatei(datei, in);
    }
}
```

Listing B.79 Datei empfangen auf der Serverseite

Im groben Ablauf des Programms ist alles, wie erwartet. Sie erzeugen einen `ServerSocket` und warten auf Verbindung. Sobald eine Verbindung eingeht, öffnen Sie den

`InputStream`, lesen den Dateinamen aus, erzeugen eine sichere Datei – gleich mehr, warum sie sicher sein muss – und schreiben die eingehenden Daten in diese Datei.

```
private String liesNameAusStream(BufferedInputStream in) throws
IOException {
    byte[] buffer = new byte[255];
    int i;
    for (i = 0; i < 255; i++) {
        int gelesen = in.read();
        if (gelesen == -1) {
            throw new IllegalStateException("Unerwartetes Ende des Datenstroms");
        }
        if (gelesen == 0) {
            break;
        }
        buffer[i] = (byte) gelesen;
    }
    return new String(buffer, 0, i, ISO_8859_1));
}
```

Listing B.80 Den Dateinamen auslesen

Hier werden, entgegen früheren Empfehlungen, Daten Byte für Byte aus einem Stream gelesen. Da es sich aber um einen `BufferedInputStream` handelt, leidet die Performance nicht darunter, und es ist so die einzige Möglichkeit, sicherzustellen, dass nur der Dateiname gelesen wird und nicht auch der Anfang des Inhalts. Sobald ein Byte mit dem Wert 0 gelesen wird, ist der Dateiname komplett, und aus den gelesenen Bytes wird ein String erzeugt. Es wäre theoretisch, durch einen Fehler auf der Clientseite, möglich, dass der Datenstrom beendet wird, bevor der Dateiname komplett ist. Das wäre daran zu erkennen, dass eine –1 aus dem Strom gelesen wird. Es gibt dann keine Chance, das Problem zu korrigieren. Das Serverprogramm wird mit einer Exception beendet. Als Nächstes wird aus dem Namen eine Datei erzeugt:

```
private File erzeugeSichereDatei(String dateiname, File verzeichnis) throws
IOException {
    File datei = new File(dateiname);
    datei = new File(verzeichnis, datei.getName());
    datei.createNewFile();
    return datei;
}
```

Listing B.81 Datei auf dem Server erzeugen

Sobald Sie eine Netzwerkverbindung akzeptieren, wird Sicherheit zum Problem. Jede offene Netzwerkverbindung kann missbraucht werden, und vor Missbrauch zu schützen, ist eine wichtige Aufgabe des Programmierers. Ein einfach auszunutzendes Problem mit dem übergebenen Dateinamen ist, dass jemand auch böswillig einen Pfad übergeben könnte und so aus dem Zielverzeichnis ausbrechen. Das wird hier verhindert, es wird explizit nur der Namensanteil des übergebenen Pfades verwendet. Wenn es nur ein Dateiname war, ändert sich dadurch nichts, wenn es aber ein Pfad mit Verzeichnisangabe war, dann wird nur der Dateiname verwendet, und die Verzeichnisse werden ignoriert. Bleibt nur noch, die Daten aus dem Strom in diese neue Datei zu schreiben. Der Code dafür entspricht genau dem, mit dem der Client in den Socket schreibt:

```
private void schreibeDatei(File datei, BufferedInputStream in) throws
FileNotFoundException, IOException {
    try (OutputStream out = new BufferedOutputStream(
     new FileOutputStream(datei))) {
        byte[] buffer = new byte[4096];
        int gelesen = 0;
        while ((gelesen = in.read(buffer)) != -1) {
            out.write(buffer, 0, gelesen);
        }
    }
}
```

Listing B.82 Aus dem Socket in die Datei schreiben

Und damit ist das Programm komplett. Jeder Teil für sich ist nicht komplex, aber zusammen übertragen sie eine Datei über eine Netzwerkverbindung – alles andere als eine triviale Aufgabe, die Sie damit gemeistert haben.

Lösung zu 13.1.2: Multithreaded Server

Schon die erste Klasse, der Client, startet mehrere Threads. Das ist nicht sonderlich komplex, aber notwendig, um beide Server richtig testen zu können. Sie werden im Code keine Überraschungen finden, es ist alles wie gehabt, mit einigen neuen Klassen:

```
public class Client implements Runnable{

    private final String hostname;
    private final int port;
    private final int id;
```

Auf einen Blick

Inhalt

5 Klassen und Objekte 125

6 Objektorientierung 155

7 Unit Testing 189

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Kai Günster ist Experte für Java-Technologien in verteilten Webanwendungen, HTML und JavaScript. Seine Projekterfahrung als Softwareentwickler reicht von E-Government über komplexe Reisereservierungssysteme bis zur IP-Telefonie. Dabei bleibt er der Java-Plattform schon seit vielen Jahren treu, lotet immer wieder gern neue Features aus und setzt HTML5 für komfortable Web-GUIs ein. Er ist Autor eines Online-Magazins und hat zwei Fachbücher geschrieben. Seine Bücher werden für ihre klare Sprache, ihren Unterhaltungswert und ihre kompakten, lehrreichen Beispiele geschätzt.

Kai Günster

Einführung in Java

734 Seiten, gebunden, 2. Auflage, August 2017
29,90 Euro, ISBN 978-3-8362-4095-6

www.rheinwerk-verlag.de/4096