Leseprobe

Sie studieren Informatik oder wollen sich beruflich in Java weiterbilden? Dann ist dieses Buch Ihre erste Wahl. Kai Günster führt Sie Schritt für Schritt in die Java-Entwicklung ein. Außerdem erhalten Sie das vollständige Inhalts- und Stichwortverzeichnis.

Kai Günster

Einführung in Java

678 Seiten, gebunden, Januar 2015
29,90 Euro, ISBN 978-3-8362-2867-1

www.rheinwerk-verlag.de/3601

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, und eine Entwicklungsumgebung, um schnell und einfach auf der Plattform Java zu 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 heute, im November 2014, 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.langpop.com*, sowie *http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html*). 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**, wird in Form von **Java Applets** aber auch im Webbrowser unterstützt. 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 auch 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 wird in Kapitel 5, »Klassen und Objekte«, und Kapitel 6, »Objektorientierung«, eingehend beleuchtet.

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 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 eben diesen Zucker doch zu bieten. In der aktuellen Version 8 von Java hält eine Erweiterung Einzug in die Sprache, nach der die Entwickler-Community seit Jahren gefleht hat: 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.

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 ausgeschlossen, aber fast. Details zum Garbage Collector finden Sie in Kapitel 17, »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 für mehrere. *Die* Implementierung der Laufzeitumgebung ist natürlich die von Oracle (ursprünglich Sun Mircosystems, 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 diese 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 8. 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 Smart Phones 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), um die Lücke zu füllen. 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 man nicht mitreden möchte, lohnt sich ein Blick in die offenen JSRs (*Java Spe-*

cification Requests), um sich über die Zukunft von Java zu informieren. So wurde zum Beispiel in JSR-337 (*https://www.jcp.org/en/jsr/detail?id=337*) schon 2010 diskutiert, was es in der 2014 fertiggestellten Version Java SE 8 Neues geben wird.

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) und *Reflection* (siehe Kapitel 4), 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).
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). Außerdem enthielt diese Version weitreichende Erweiterungen an der Klassenbibliothek, unter anderem Unterstützung für reguläre Ausdrücke (siehe Kapitel 8) 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) ► *Annotationen* (siehe Kapitel 6) ► *Enumerations* (Typen mit aufgezählten Werten, siehe Abschnitt 6.5) ► *Varargs* (variable Parameterlisten, siehe Kapitel 10) ► 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). 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	Voraussichtlich 2016	Für die nächste Java-Version ist unter anderem ein Modulsystem im Gespräch und, darauf basierend, die Modularisierung der Klassenbibliothek. So würde Java nicht mehr mit mehreren tausend Klassen ausgeliefert, stattdessen würden benötigte Module bei Bedarf nachgeladen.
Java 10	2018?	Über diese Version kursieren Gerüchte, dass Primitivtypen (siehe Kapitel 2) 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, um Code umzustrukturieren und sich wiederholenden Code zu generieren.

In diesem Buch wird die Netbeans-Entwicklungsumgebung verwendet, 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 entweder in den Downloads zum Buch oder können 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), in dem auch die Entwicklerwerkzeuge enthalten sind. 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 8 herunterladen.

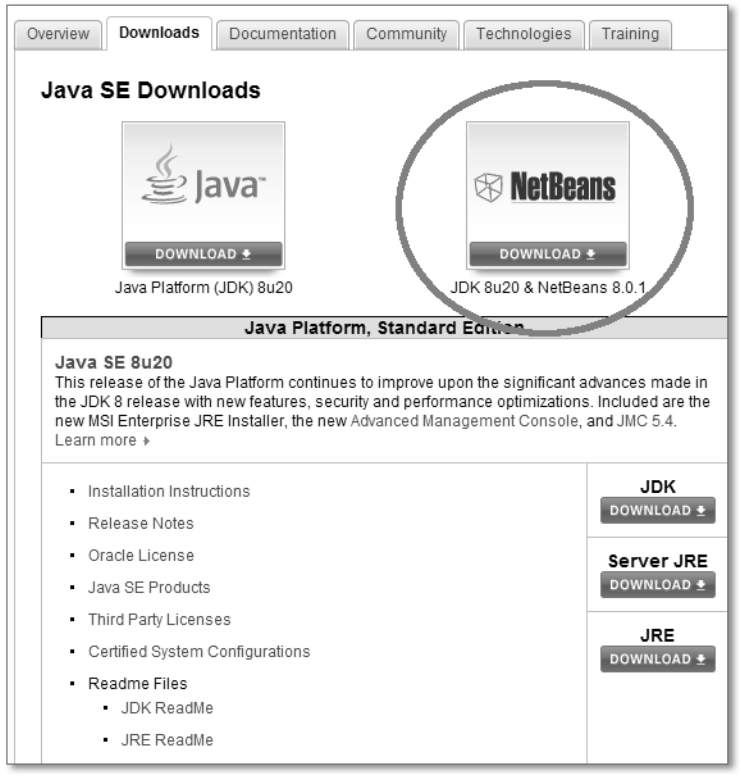

Abbildung 1.1 Java-Download von der Oracle-Webseite

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.

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.

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

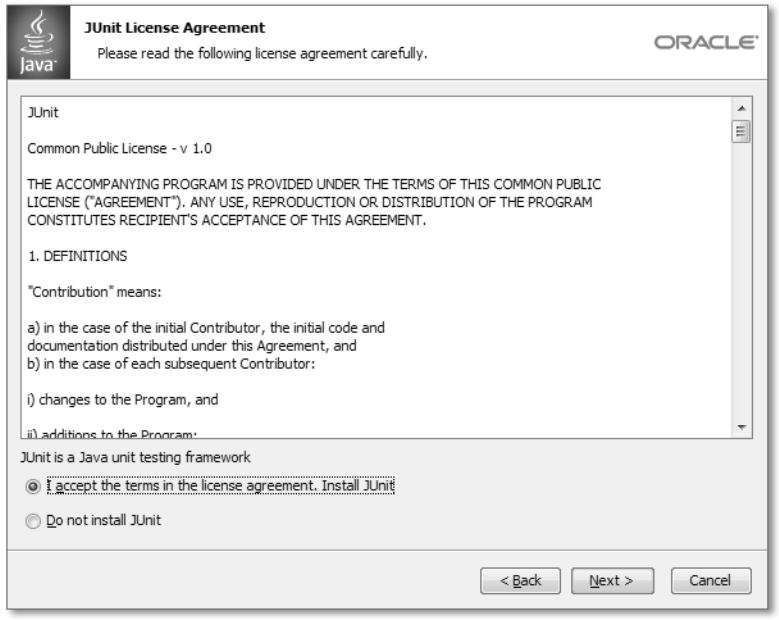

Abbildung 1.2 JUnit installieren

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.

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.

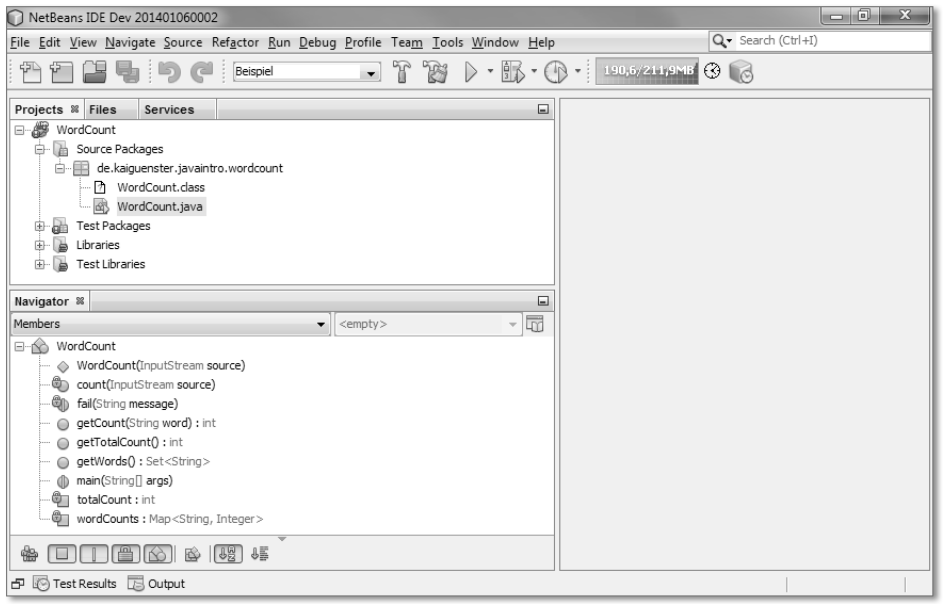

Abbildung 1.3 Netbeans: Projekt geöffnet

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 (siehe Abbildung 1.4).

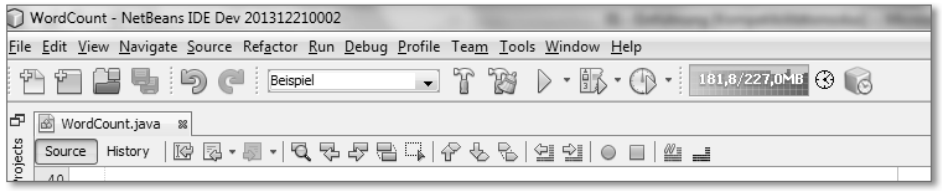

Abbildung 1.4 Das WordCount-Programm ausführen

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.

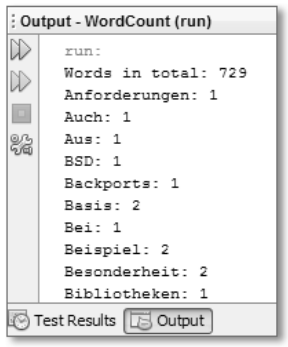

Abbildung 1.5 Die Ausgabe von WordCount

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 wird im Laufe des Buches im Detail erklärt.

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 genau 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 man mehrere Klassen des gleichen Namens aus unterschiedlichen Packages verwenden muss, denn diese 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). Maps 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 `total-`

`Count` 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 werden in Kapitel 10, »Arrays und Collections«, erläutert).

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){
    count(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, Punktation, 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 hinzuaddiert, 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.

Weiterhin 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 mithilfe von `count.getTotalCount()` die Gesamtwortzahl ausgegeben , dann in alphabethischer Reihenfolge die Vorkommen einzelner Wörter. Für die Ausgabe der einzelnen Wörter werden die neue Stream-API von Java 8 sowie die neuen Lambda-Ausdrücke (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 Pro-

gramms 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, ist es nötig, dass 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 Befehlt 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, das 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 beinhaltet einen Algorithmus, um Wortvorkommen in einem Text zu zählen. »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("Fibonaccizahlen 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 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 vorne. 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 wir 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 diesen 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 8 enthält mehr als 4.000 Klassen. Glück-

licherweise ist die Klassenbibliothek in Packages unterteilt. Wenn Sie die wichtigsten Packages und ihre Aufgaben kennen, so haben Sie dadurch einen guten Ausgangspunkt, um 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), aber auch die grundlegenden Datentypen wie `String` und die verschiedenen Arten von `Number` (`Integer`, `Float`, ...). Ebenfalls hier zu finden ist die Klasse `Thread`, die parallele Programmierung in Java ermöglicht (siehe Kapitel 13), 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 Unterpackages sind eine Sammlung von diversen Werkzeugen, die in vielen Programmen nützlich sind. Dazu gehören zum Beispiel Listen und Mengen, ein Zufallszahlengenerator, `Scanner` und `StringTokenizer`, um eine Zeichenkette zu zerlegen, und vieles mehr. In Unterpackages finden Sie unter anderem reguläre Ausdrücke (`java.util.regex`), Werkzeuge zum Umgang mit ZIP-Dateien (`java.util.zip`) und eine Werkzeugsammlung, um konfigurierbare Logdateien zu schreiben (`java.util.logging`). Viele Klassen aus diesem Package werden Sie im Laufe des Buches kennenlernen, vor allem in Kapitel 8.
`java.applet`	Hier finden Sie Klassen, um Java-Applikationen in Webseiten einzubinden, die bekannten Java Applets.
`java.awt`	`java.awt` und seine Unterpackages 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.

Tabelle 1.2 Die wichtigsten Packages der Java 8 Standard Edition

Package	Inhalt
`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, um Daten aus anderen Quellen zu lesen bzw. dorthin zu 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. 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, um Kommunikation per UDP oder TCP als Client oder als Server zu 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.

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

Package	Inhalt
java.sql	Dieses Package dient der Anbindung von Java-Anwendungen an SQL-Datenbanken wie MySQL. Man benötigt dazu einen Datenbanktreiber, der inzwischen von allen Datenbankherstellern zur Verfügung gestellt wird, und kann 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, um mit Zeit- und Datumswerten zu arbeiten sowie zum Umgang mit Zeitzonen.
javax.swing	Das zweite Package, um grafische Benutzeroberflächen zu erstellen. 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 in Kapitel 16 ausführlich besprochen wird.

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 Hauptpackage javax, dessen Einsatzgebiet aber spezieller ist. Auch haben viele der aufgeführten Packages wiederum Unterpackages. Eine komplette Übersicht über alle verfügbaren Packages und Klassen bietet Ihnen die aktuelle Version des *Javadoc*.

1.7 Dokumentieren als Gewohnheit – Javadoc

Sie haben Javadoc oben bereits kennengelernt, um 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 8 unter der URL *http://download.java.net/jdk8/docs/api/*.

1.7.1 Den eigenen Code dokumentieren

Wie Sie in den Beispielen auch 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.

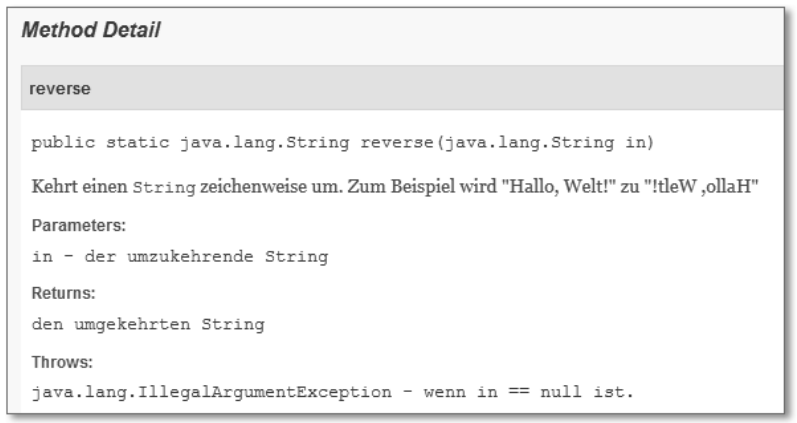

Abbildung 1.7 Die generierte HTML-Dokumentation

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) 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 haben 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 auch noch 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. *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.

Tabelle 1.5 Allgemeine Javadoc-Tags

Tag	Bedeutung
`@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)`, um einen Link auf die Methode `reverse` der Klasse `Reverse` zu erzeugen. 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 entfällt 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 allem 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, um zu steuern, aus welchen Quelldateien Dokumentation erzeugt werden soll und wie die Ausgabe aussieht. Diese 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 mit der 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 mit Javadoc versehen werden?

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 ZIPs 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 von 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 erzeugten 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 werden soll (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:

```
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 beinhalten 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.

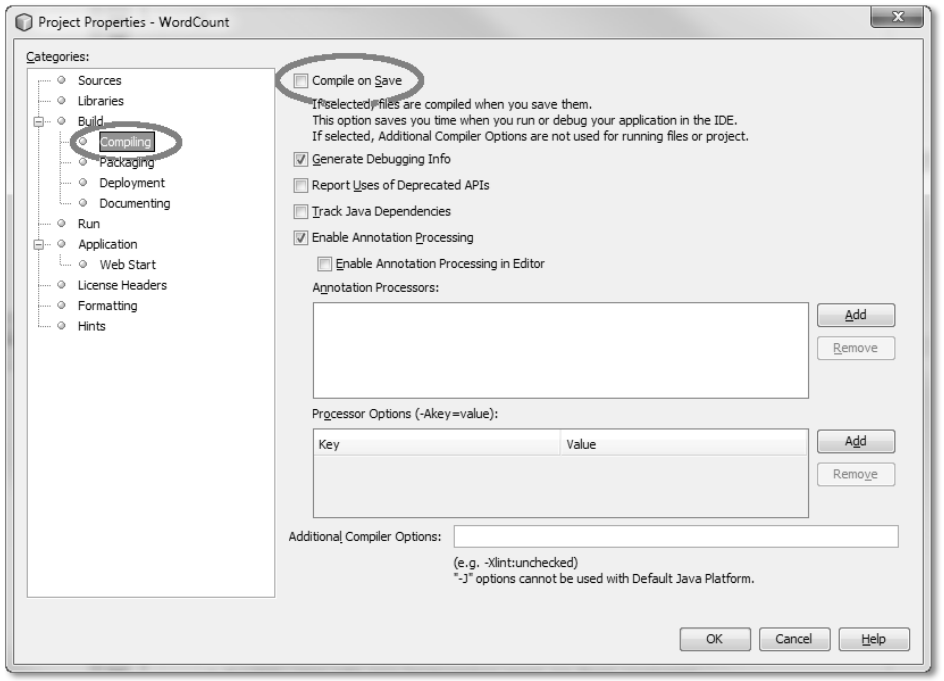

Abbildung 1.8 »Compile on Save« deaktiviert

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.

1.9.2 Breakpoints und schrittweise Ausführung

Ein Breakpoint (Unterbrechungspunkt) ist eine Stelle im Code, an der die Ausführung des Programms angehalten wird. Sie können einen Breakpoint setzen, 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 dem nächsten Breakpoint.
- Step Over: Die aktuelle Zeile des Programms, in Netbeans zu erkennen am grünen Hintergrund, wird ausgeführt. In der nächsten Zeile wird es 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 Callstack 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.

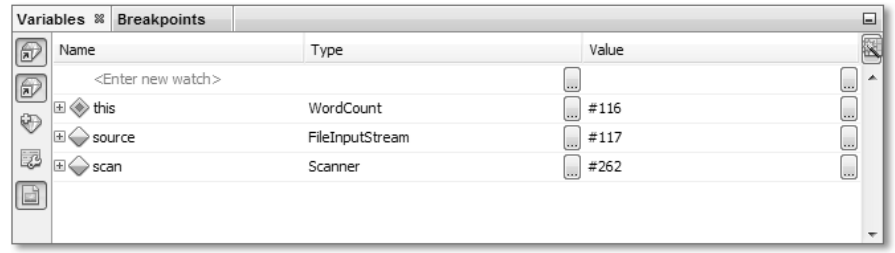

Abbildung 1.10 Die Variablenansicht des Debuggers

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).

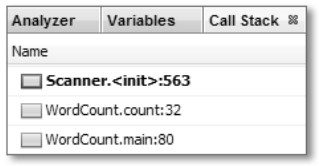

Abbildung 1.11 Die Stackansicht des Debuggers

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, zum anderen um 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).

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 man einmal berechnete Werte in einem Array speichert und sie beim zweiten Mal von dort

liest, anstatt sie erneut zu berechnen. Dadurch würde aber der Algorithmus weniger klar und verständlich.

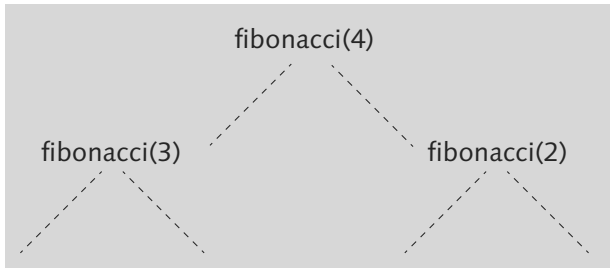

Abbildung 1.12 Rekursive Aufrufe von »fibonacci«

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.

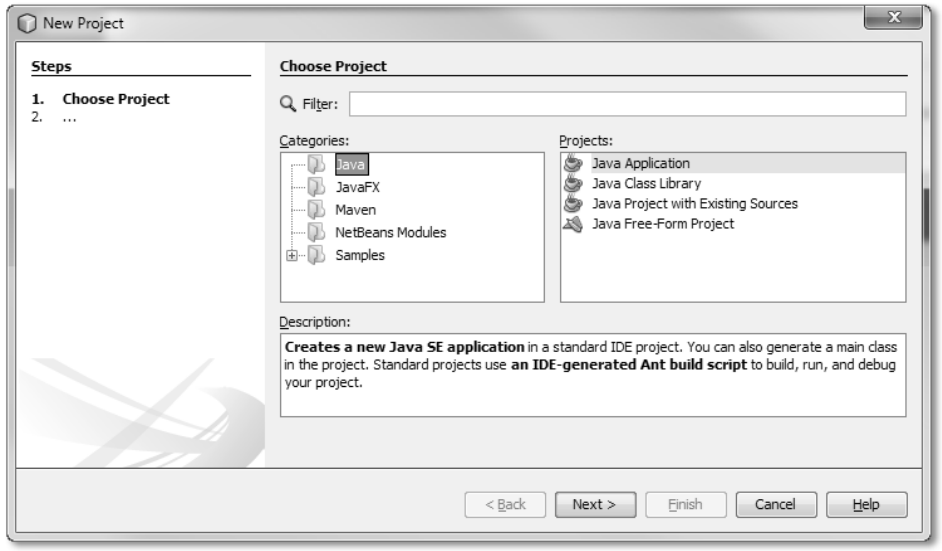

Abbildung 1.13 Neues Projekt anlegen

Der Dialog, um ein neues Projekt anzulegen, 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 diesen aus, und klicken Sie auf NEXT (siehe Abbildung 1.13).

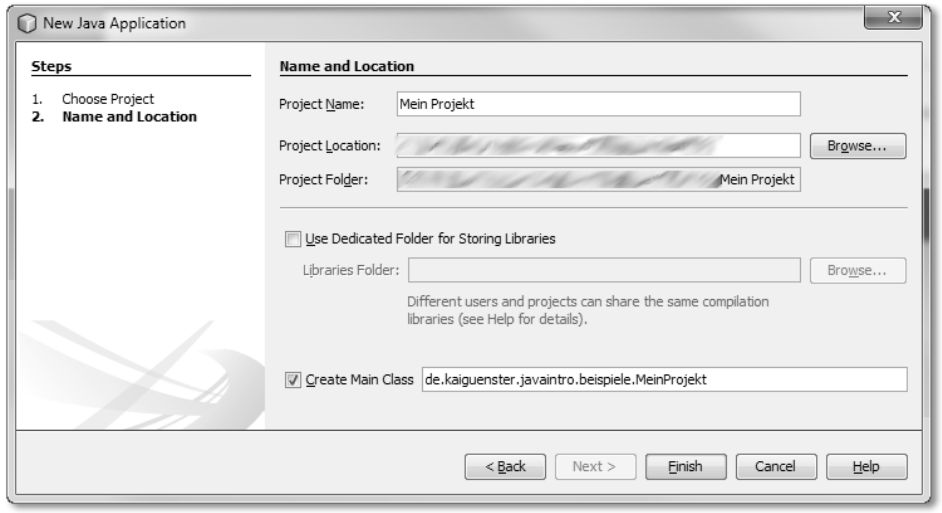

Abbildung 1.14 Neues Project anlegen

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.

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 2

Variablen und Datentypen

In jedem Programm geht es auf die eine oder andere Art um die Verarbeitung von Daten. Eine wichtige Voraussetzung dazu ist, Daten zunächst im Speicher halten zu können. Dazu dienen Variablen. In diesem Kapitel werden Sie lernen, wie Sie Variablen deklarieren und mit ihnen arbeiten. Außerdem werden Sie die sogenannten primitiven Datentypen kennenlernen und wie Sie mit ihnen einfache Berechnungen durchführen sowie den Unterschied zwischen primitiven Datentypen und Objekttypen.

Bevor Sie mit einem ersten Programm loslegen können, müssen Sie wissen, wie Sie mit Daten arbeiten, schließlich ist es der Zweck eines jeden Programms, Daten zu verarbeiten.

2.1 Variablen

In Kapitel 1 haben Sie bereits kurz gelesen, was Variablen sind: benannte Speicherstellen, in denen Sie Ergebnisse von Berechnungen und Methodenaufrufen speichern können. Der Nutzen ist offensichtlich, viele Algorithmen sind ohne ein »Gedächtnis« nicht umzusetzen.

Variablennamen

Variablennamen, genau wie Methoden- und Klassennamen, dürfen beliebige Unicode-Buchstaben und Ziffern enthalten, ebenso wie die Zeichen _ und $. Das erste Zeichen darf dabei keine Ziffer sein. Per Konvention werden aber nur Buchstaben und Ziffern verwendet, und das erste Zeichen ist immer ein Kleinbuchstabe. Zum Beispiel `summand1`, `summe` usw.

Leerzeichen sind in Variablennamen nicht gestattet. Soll ein Name aus mehreren Wörtern bestehen, wird dies durch *CamelCasing* umgesetzt: Leerzeichen werden einfach ausgelassen, die Anfangsbuchstaben aller Wörter außer dem ersten werden großgeschrieben. Zum Beispiel: `summeGesamt`, `meinAuto` usw.

> Sie dürfen für Bezeichner zwar beliebige Unicode-Buchstaben verwenden, empfehlenswert ist aber, sich auf ASCII-Zeichen zu beschränken. So vermeiden Sie Schwierigkeiten, wenn Ihr Code in einer anderen Umgebung bearbeitet werden soll.

Jede Variable in Java hat zwei definierende Eigenschaften, ihren Namen und ihren Typ. Der Variablenname ist ein fast beliebiger Bezeichner, mit dem Sie auf die Variable zugreifen. Der Typ definiert, welche Art von Daten in dieser Variablen gespeichert wird. Bevor eine Variable benutzt werden kann, muss sie deklariert werden, zum Beispiel so:

```
int zahl;
```

Der Name dieser Variablen ist `zahl`, ihr Datentyp ist `int`, also eine Ganzzahl. Mehr zu den verschiedenen Datentypen folgt im Laufe dieses Kapitels. Wenn eine Variable deklariert ist, können Sie ihr einen Wert des richtigen Typs zuweisen und die Variable überall verwenden, wo Sie auch ihren Wert direkt verwenden könnten.

```
int summand1;
summand1 = 5;
int summand2 = 17;
int summe = summand1 + summand2;
```

Listing 2.1 Einfache Addition mit drei Variablen

Dieses kurze Codefragment zeigt sowohl, wie Sie einer Variablen einen Wert zuweisen, als auch, wie Sie mit diesem Wert weiterrechen. In den ersten beiden Zeilen sehen Sie, wie die Variable `summand1` deklariert und ihr anschließend ein Wert zugewiesen wird. Zeile 3 tut das Gleiche für die Variable `summand2`, Deklaration und Zuweisung sind lediglich in einer Zeile zusammengefasst. In der letzten Zeile werden die Werte der beiden Summanden addiert und das Ergebnis dieser Addition in der neuen Variablen `summe` gespeichert.

Es ist auch möglich, mehrere Variablen gleichen Typs in einer Zeile zu deklarieren.

```
int zahl1, zahl2 = 5, zahl3;
```

Die Zuweisung gilt dabei nur für die Variable, für die sie auch ausgeschrieben wird: `zahl2` hat den Wert 5.

Ob `zahl1` und `zahl3` nach der Deklaration bereits einen Wert haben, hängt davon ab, wo sich die Deklaration befindet. Felder, also Variablen, die zu einem Objekt gehören, erhalten den Default-Wert 0. Lokale Variablen, also solche, die innerhalb einer Methode deklariert werden, haben keinen Default-Wert. Bevor Sie mit einer lokalen

Variablen arbeiten, müssen Sie ihr einen Wert zuweisen, sonst meldet der Compiler den Fehler: »Variable might not have been initialized.«

2.1.1 Der Zuweisungsoperator

Es ist wichtig, zu verstehen, dass sich das Gleichheitszeichen als *Zuweisungsoperator* vom Gleichheitszeichen, das Sie aus der Mathematik kennen, unterscheidet. Das mathematische Gleichheitszeichen sagt aus, dass die linke Seite und die rechte Seite gleich sind. Der Zuweisungsoperator schreibt den Wert von der rechten Seite in die Variable auf der linken Seite. Dabei ist er nicht symmetrisch, auf der linken Seite muss eine – und nur eine – Variable stehen, auf der rechten Seite der Wert, der in dieser Variablen gespeichert werden soll. Es ist deshalb falsch zu schreiben `17 = int zahl`, der Compiler lehnt dies mit einer Fehlermeldung ab. Durch diese Asymmetrie ist auch die Bedeutung des folgenden Codefragments eindeutig. Welchen Wert hat am Ende die Variable `zahl1`?

```
int zahl1 = 7;
int zahl2 = 29;
zahl1 = zahl2;
```

Listing 2.2 Asymmetrie des Zuweisungsoperators

Die richtige Antwort lautet `29`. In der letzten Zeile steht `zahl1` auf der linken Seite, so wird ihr ein neuer Wert zugewiesen. Auf der rechten Seite steht ebenfalls eine Variable, ihr Wert (`29`) wird ausgelesen und in der Variablen links gespeichert.

2.1.2 Scopes

Neben Name und Typ gibt es eine dritte Eigenschaft für Variablen, die Sie nicht explizit deklarieren, die aber dennoch sehr wichtig ist: den *Scope*. Der Scope einer Variablen legt ihre Sichtbarkeit fest, also von wo aus auf sie zugegriffen werden kann. Die Sichtbarkeit von Klassen- und Instanzvariablen wird uns erst in Kapitel 5 beschäftigen, aber auch *lokale Variablen*, also solche, die innerhalb einer Methode definiert werden, haben einen Scope. In den Beispielen des vorigen Kapitels haben Sie bereits gesehen, dass Sie mit geschweiften Klammern Codeblöcke definieren. Die geschweiften Klammern der Methodendeklaration umschließen einen Block, die geschweiften Klammen um den Rumpf eines `if`-Konstrukts umschließen einen Block. Daran sehen Sie bereits, dass Blöcke ineinander verschachtelt werden können.

Auf eine Variable zugreifen können Sie innerhalb des Blocks, in dem sie deklariert wurde, sowie in allen darin verschachtelten Blöcken.

```
public Zeit liesZeit(){
    int stunden = liesStunde();
    if (stunden != null){
        int minuten = liesMinute();
        return new Zeit(stunden, minuten);
    }
    return new Zeit(stunden, minuten); //<-Compilerfehler!!!
}
```

Listing 2.3 Variablenscopes

Im Beispiel ist die Variable `stunden` in der ganzen Methode sichtbar, `minuten` aber nur innerhalb des `if`-Blocks. Der Versuch, in der letzten Zeile der Methode auf `minuten` zuzugreifen, egal, ob lesend oder schreibend, führt zu einem Compilerfehler.

2.1.3 Primitive und Objekte

Es gibt in Java zwei Gruppen von Typen, die Sie zwar im Wesentlichen gleich behandeln können, zwischen denen es aber Unterschiede gibt. Das ist zum einen die offene Gruppe der Objekttypen. Offen deshalb, weil Sie neue Objekttypen definieren können und werden (mehr dazu folgt in Abschnitt 2.3, »Objekttypen«). Zum anderen gibt es die geschlossene Gruppe der Primitivtypen.

2.2 Primitivtypen

Die Primitivtypen in Java bilden eine geschlossene Gruppe, das heißt, es ist nicht möglich, neue Primitivtypen zu definieren. Es gibt acht Typen und nicht mehr. Dies sind die verschiedenen numerischen Typen `byte`, `short`, `int`, `long`, `float` und `double`, der Zeichentyp `char` und der Wahrheitswert `boolean`.

Die Zukunft der Primitivtypen

Die Primitivtypen stellen einen Makel in Javas Objektorientierung dar, denn Primitive sind keine Objekte. Durch sie kann Java nicht als rein objektorientierte Sprache angesehen werden.

Sie werden in Abschnitt 2.4.3, »Implizite Konvertierung«, sehen, dass die Grenzen zwischen Primitivtypen und Objekttypen bereits verwischt sind. Vor diesem Hintergrund ist es nicht ausgeschlossen, dass Primitivtypen in Java 10 völlig abgeschafft werden, wie Gerüchte es behaupten. Ob es aber wirklich so kommt und wie das dann im Detail umgesetzt wird, bleibt abzuwarten.

2.2.1 Zahlentypen

Sechs der acht Primitivtypen sind numerische Typen. Vier von ihnen sind ganzzahlig, zwei für Dezimalzahlen. Sie unterscheiden sich jeweils in ihrem Wertebereich.

Ganzzahlige Typen

Die verschiedenen ganzzahligen Typen unterscheiden sich in den Werten, die sie aufnehmen können, und im Speicher, den sie belegen (siehe Tabelle 2.1). In einem modernen Computer, der über mehrere Gigabyte Speicher verfügt, scheint es unnötig, sich über den Unterschied zwischen 1, 2, 4 oder sogar 8 Byte Gedanken zu machen. Selbst ein handelsübliches Smartphone hat zurzeit 2 GB Speicher, welchen Sinn hat es noch, ein `byte` anstelle eines `int` zu verwenden?

Typ	Belegter Speicher	Minimalwert	Maximalwert
`byte`	8 Bit/1 Byte	–128	127
`short`	16 Bit/2 Byte	–32.768	32.767
`int`	32 Bit/4 Byte	–2.147.483.648	2.147.483.647
`long`	64 Bit/8 Byte	-2^{63}	$2^{63}-1$

Tabelle 2.1 Ganzzahlige Typen und ihre Wertebereiche

Obwohl Speicher in den letzten Jahren preiswert geworden ist, sind die verschiedenen Zahlentypen kein Relikt aus der Vergangenheit, es gibt nach wie vor gute Gründe, sie zu verwenden:

- Computer und Smartphones sind nicht die einzigen Umgebungen, in denen Java ausgeführt wird. Java-ME- und Embedded-Java-Anwendungen werden auf Hardware ausgeführt, auf denen Speicher viel stärker begrenzt ist.
- Auch wenn es für eine einzelne Variable selten einen Unterschied macht, ob sie 1 Byte belegt oder 4, so sieht es doch ganz anders aus, wenn Sie viele solcher Variablen haben. Ein maximal großes Array in Java hat 2.147.483.647 Einträge. Das sind für ein `byte`-Array 2 GB, für ein `int`-Array 8 GB. Es ist zwar nicht alltäglich, mit Arrays in dieser Größe umgehen zu müssen, aber es kommt vor.
- Es ist in Java sehr einfach, Objekte über ein Netzwerk zu verschicken. Auch wenn Speicherverbrauch kein Problem darstellt, ist Datenübertragung immer noch ein Engpass, den Sie durch passende Datentypen etwas entschärfen können, vor allem in Verbindung mit dem vorigen Punkt.

Alle vier ganzzahligen Typen können Sie, wie Sie in den Beispielen bereits gesehen haben, im Code als Literal verwenden – eine umständliche Art, auszudrücken, dass Sie Zahlen einfach in den Code schreiben können:

```
int summe = 145355 + 67443355;
```

Listing 2.4 Addition von zwei »int«-Literalen

Literale haben immer den Typ `int`, können aber ohne Probleme auch den anderen Zahlentypen zugewiesen werden, wenn sie in deren Wertebereich liegen. Wenn Sie versuchen, ein unpassendes Literal einer Variablen zuzuweisen, bemängelt dies der Compiler. Um ein `long`-Literal zu erzeugen, dessen Wert über den Wertebereich von `int` hinausgeht, müssen Sie dieses explizit anfordern, indem Sie ein `l` oder `L` der Zahl anhängen:

```
long grosseZahl = 10000000001000000000L;
```

Normalerweise geben Sie Zahlenwerte im Dezimalsystem an. In manchen Situationen ist es aber sinnvoller und verständlicher, ein anderes Zahlensystem zu verwenden. In Java können Sie Zahlen im Binär- und im Hexadezimalsystem verwenden, indem Sie sie mit dem Präfix `0b` bzw. `0x` versehen. Für Berechnungen mit dem Wert macht das natürlich keinen Unterschied, es dient lediglich der Verständlichkeit des Codes.

```
//Binär
int binaer = 0b1001101;
//Dezimal
int dezimal = 17455;
//Hexadezimal
int hex = 0x5FE;
```

Listing 2.5 »int«-Literale in verschiedenen Zahlensystemen

Dezimalzahlen

Genau wie die ganzzahligen Typen unterscheiden sich die Zahlentypen mit Fließkomma, `float` und `double`, in Speicherbedarf und Wertebereich (siehe Tabelle 2.2). Sie unterscheiden sich aber auch in ihrer Genauigkeit, also in der Anzahl der Nachkommastellen, die sie zuverlässig darstellen können. Durch die interne Darstellung dieser Typen (für die Interessierten, es handelt sich um Fließkommazahlen nach dem Standard IEEE 754 mit einfacher [`float`] bzw. doppelter [`double`] Genauigkeit) lassen sich nicht alle Werte exakt darstellen, was zu mathematischen Unannehmlichkeiten führt. Mit `double` berechnet ist zum Beispiel 0,3 – 0,1 = 0,19999999999999998.

Typ	Belegter Speicher	Kleinster positiver Wert	Maximalwert
`float`	32 Bit/4 Byte	$1{,}4 \cdot 10^{-45}$	$3{,}4 \cdot 10^{38}$
`double`	64 Bit/8 Byte	$4{,}9 \cdot 10^{-324}$	$1{,}7 \cdot 10^{308}$

Tabelle 2.2 Dezimale Typen und ihre Wertebereiche

Und um alles noch verwirrender zu machen, ist die Anzahl der korrekten Nachkommastellen nicht für alle Werte gleich, manche Zahlen lassen sich von `float` und `double` genauer annähern als andere. Für viele Anwendungen ist die Ungenauigkeit aber nicht relevant, da es immer nur zu sehr geringen Abweichungen kommt. Trotzdem sollten Sie einige wichtige Punkte beachten.

- Um einen `float`- oder `double`-Wert auszugeben, sollten Sie ihn immer mit der passenden Anzahl an Nachkommastellen formatieren. Das geht sehr einfach mit der Klasse `java.text.DecimalFormat` (siehe Kapitel 8, »Die Standardbibliothek«) und vermeidet Ausgaben wie: »Fügen Sie 199,98639552124533 g Butter hinzu.«
- Prüfen Sie `float`- und `double`-Werte niemals mit dem `==`-Operator auf Gleichheit. Die Bedingung von `if (einDouble == 3)` kann unter Umständen nicht erfüllt werden, denn der Wert von `einDouble` könnte als Ergebnis einer Berechnung 2,99999999999 sein. Um die `if`-Bedingung korrekt umzusetzen, ist zu prüfen, ob die Differenz der beiden Werte unterhalb einer Grenze liegt: `if (Math.abs(einDouble - 3) < 0.00001)`. Die Methode `Math.abs` ermittelt den Betrag des übergebenen Parameters, so ist es egal, ob die Differenz positiv oder negativ ist.
- Wenn Sie auf genaue Ergebnisse angewiesen sind und sich keine Rundungsfehler erlauben können, benutzen Sie statt `float` oder `double` die Klasse `BigDecimal`. `BigDecimal`-Objekte belegen zwar mehr Speicher als ein `double`-Wert, und Berechnungen mit ihnen sind weniger performant, dafür können sie aber mit beliebig vielen Nachkommastellen fehlerfrei umgehen.

Die lange Liste an Warnungen sollte Sie aber nicht abschrecken, Fließkommatypen zu benutzen. In den meisten Fällen, in denen Sie mit Dezimalzahlen umgehen müssen, ist `double` dennoch die richtige Wahl. Wenn es keinen guten Grund gibt, das speichersparendere `float` zu verwenden, sollten Sie auch immer die genaueren `double`-Werte vorziehen. Diesem Grundsatz entsprechend sind auch Fließkommaliterale in Java vom Typ `double`, wenn nicht anders angegeben. Für ein `float`-Literal ist ein `f` oder `F` der Zahl anzuhängen. In beiden Fällen ist das Trennzeichen für die Dezimalstellen der Punkt, wie im englischsprachigen Raum üblich, nicht das Komma.

```
double d = 0.1234567;
float f = 0.1234567f;
```

Listing 2.6 »float«- und »double«-Literale

Typumwandlung

Wie Sie oben bereits gesehen haben, können Sie ein Literal, das formal vom Typ `int` ist, einer Variablen vom Typ `short` oder `byte` zuweisen, wenn es in ihren Wertebereich fällt. Das ist nicht möglich, wenn der zuzuweisende Wert in einer Variablen steht. Selbst wenn der Wert in die Zielvariable passen würde, kann der Compiler dies nicht mit Sicherheit bestimmen und gibt für den folgenden Code eine Fehlermeldung aus:

```
int i = 1;
short s = i;
```

Listing 2.7 Fehlerhafte Zuweisung an eine »short«-Variable

Andersherum ist eine Zuweisung von `short` nach `int` möglich, da jeder `short`-Wert auch in einer `int`-Variablen Platz findet. Auf das gleiche Problem treffen Sie auch zwischen `float` und `double`: Sie können den Inhalt einer `float`-Variablen einer `double`-Variablen zuweisen, nicht jedoch umgekehrt.

Es gibt aber die Möglichkeit, diese Einschränkung zu umgehen, indem Sie eine *Typumwandlung* (engl. *cast*) erzwingen. Dazu geben Sie nur den Zieltyp in Klammern vor der Quellvariablen an:

```
int i = 1;
short s = (short)i;
```

Listing 2.8 Zuweisung mit Cast

So gibt es keine Compilerfehler mehr, die Zuweisung wird problemlos durchgeführt. Es findet allerdings keine Prüfung statt, ob die Variable den neuen Wert aufnehmen kann, weder durch den Compiler noch durch die Laufzeitumgebung. Das führt zu interessanten und schwer zu findenden Fehlern. Was ist zum Beispiel die Ausgabe des folgenden Codefragments?

```
int i = 1000000000;
short s = (short) i;
System.out.println(s);
```

Listing 2.9 Cast mit unpassendem Wertebereich

Dieser Code gibt die Zahl –13824 aus. Der Grund dafür ist alles andere als offensichtlich. Bei der Umwandlung von `int` (4 Byte) nach `short` (2 Byte) werden die höherwertigen 2 Byte des `int`-Wertes verworfen. Dass das Ergebnis dieser Operation eine negative Zahl ist, liegt an der internen Darstellung von negativen Zahlen: Ist das höchste Bit einer Zahl auf 1 gesetzt, wird die Zahl als negative Zahl (im Zweierkomplement) interpretiert. War das 16. Bit des `int`-Wertes eine 1, so ist der `short`-Wert nach

dem Cast eine negative Zahl. Deshalb sollten Sie sich über den Wertebereich sicher sein, wenn Sie Zahlen casten.

Auf die gleiche Art funktioniert auch die Umwandlung von Dezimalzahlen und sogar von Dezimalzahlen in Ganzzahlen. Dabei werden Nachkommastellen verworfen, ohne dass negative Seiteneffekte auftreten. Auch hier gilt aber, dass der Versuch, eine Zahl außerhalb des Wertebereichs zu casten, zu unerwünschten Ergebnissen führt.

2.2.2 Rechenoperationen

Mit allen Zahlentypen können Sie die vier Grundrechenarten anwenden sowie eine weitere Berechnung, die in der Schule nicht zu den Grundrechenarten gezählt wird. Java kennt sie als Operatoren (siehe Tabelle 2.3).

Operation	Operator	Bemerkung
Addition	+	Der +-Operator wird neben der Addition auch für die String-Konkatenation verwendet, also um zwei Zeichenketten zu einer zusammenzufügen (siehe unten).
Subtraktion	-	
Multiplikation	*	
Division	/	
Modulo	%	Die Modulo-Operation gibt den Divisionsrest zurück. Zum Beispiel ist `5 % 3 = 2`. In Java ist Modulo auch für Dezimalzahlen definiert, `0.5 % 0.3` ergibt wie erwartet `0.2`.

Tabelle 2.3 Rechenoperatoren in Java

Die Anwendung der Operatoren entspricht der gewohnten Anwendung aus der Mathematik: in Infixschreibweise. Sehen Sie dazu das folgende Beispiel:

```
public static void main(String[] args) {
    …
    int zahl1 = Integer.parseInt(args[0]);
    int zahl2 = Integer.parseInt(args[1]);
    System.out.println("Summe: " + (zahl1 + zahl2));
    System.out.println("Differenz: " + (zahl1 - zahl2));
    System.out.println("Produkt: " + (zahl1 * zahl2));
```

```
    System.out.println("Quotient: " + (zahl1 / zahl2));
    System.out.println("Divisionsrest: " + (zahl1 % zahl2));
}
```

Listing 2.10 Grundrechenarten in Java

Neben der offensichtlichen Anwendung der Rechenoperatoren gibt es im Code zwei erwähnenswerte Dinge. Das erste Pluszeichen in den Ausgabezeilen ist jeweils keine Addition, sondern eine String-Konkatenation: Die Zeichenkette links wird mit dem Ergebnis der Berechnung rechts zu einer einzigen Zeichenkette verknüpft, die mit `System.out.println` ausgegeben wird.

Dadurch, dass Konkatenation und Addition in einer Anweisung verwendet werden, wird es notwendig, die Addition in Klammern zu schreiben, um die Reihenfolge der Operationen zu erzwingen. Bei den anderen Berechnungen wäre dies nicht notwendig, nur das Pluszeichen hat in Java mehrere Funktionen. Diese Klammern dienen der Einheitlichkeit und Übersichtlichkeit. Warum die Klammern bei der Addition notwendig sind, wird am Beispiel klar. Nehmen Sie für `zahl1` und `zahl2` die Werte 1 und 2 an.

Mit Klammern:

1. `"Summe: " + (1 + 2)` – es wird zunächst der Inhalt der Klammern berechnet, das Pluszeichen ist eindeutig als Addition zu erkennen.
2. `"Summe: " + 3` – ein Operand des verbleibenden Pluszeichens ist ein String, es muss sich um eine Konkatenation handeln.
3. Ausgabe: `"Summe: 3"`

Ohne Klammern:

1. `"Summe: " + 1 + 2` – die beiden Operationen Addition und Konkatenation haben dieselbe Priorität, die gesamte Anweisung wird von links nach rechts ausgewertet. Das erste Pluszeichen hat einen `String` als Operand und muss somit als Konkatenation interpretiert werden.
2. `"Summe: 1" + 2` – nun hat auch das zweite Pluszeichen einen String-Operanden, also wird erneut konkateniert.
3. Ausgabe: `"Summe: 12"`

Die zweite Auffälligkeit des Beispiels ist nicht im Code zu erkennen, fällt aber sofort auf, wenn Sie das Programm mit den richtigen Parametern ausführen. Versuchen Sie es zum Beispiel mit den Zahlen 5 und 3.

Die Ergebnisse von Addition, Subtraktion, Multiplikation und Modulo sind genau wie erwartet, aber das Ergebnis der Division überrascht: Ausgegeben wird das ganz-

zahlige Divisionsergebnis 1. Der Grund dafür ist die Art, wie Java den Ergebnistyp der Rechenoperationen bestimmt.

Typkonvertierung bei Rechenoperationen

Bei allen Rechenoperationen werden zunächst die Operanden in kompatible Datentypen umgewandelt. Die JVM geht dabei immer wie folgt vor:

- Wenn einer der Operanden vom Typ `double` ist, so wird der andere nach `double` umgewandelt.
- Ansonsten wird, wenn einer der Operanden vom Typ `float` ist, auch der andere nach `float` umgewandelt.
- Ansonsten wird, wenn einer der Operanden vom Typ `long` ist, auch der andere nach `long` umgewandelt.
- Ansonsten werden beide Operanden nach `int` umgewandelt.

Das Ergebnis der Rechenoperation hat immer den gleichen Typ, den beiden Operanden nach der Umwandlung haben. Der Grund für das unerwartete Divisionsergebnis wird so klar: Beide Operanden sind vom Typ `int`, deshalb ist auch das Ergebnis vom Typ `int`. Java rundet in diesem Fall übrigens nicht kaufmännisch, sondern verwirft die Nachkommastellen, es handelt sich um echte Ganzzahldivision.

Es gibt einen weiteren Fallstrick in diesem System von Konvertierungen, wenn Sie mit großen `int`-Werten rechnen. Betrachten Sie die Rechnung `Integer.MAX_VALUE + Integer.MAX_VALUE`. Die Konstante `Integer.MAX_VALUE` enthält den größtmöglichen Wert für eine `int`-Variable. Da beide Operanden den Typ `int` haben, hat auch das Ergebnis den Typ `int`. Offensichtlich passt der doppelte Maximalwert eines `int` nicht in einen `int`, es kommt bei der Rechnung zu einem Überlauf. Das Ergebnis lautet –2. Der Grund, warum das Ergebnis eine negative Zahl ist, ist wie auch schon bei Typumwandlungen in der internen Darstellung von Ganzzahlen zu suchen. Die Berechnung kann korrekt ausgeführt werden, wenn Sie einen der Operanden nach `long` casten: `Integer.MAX_VALUE + (long)Integer.MAX_VALUE`. Dadurch wird auch der Ergebnistyp `long` und kann das Rechenergebnis aufnehmen. Es reicht nicht aus, erst das Ergebnis zu casten, denn dann wurde es bereits mit einem Überlauf berechnet, und das falsche Ergebnis wird auf den richtigen Datentyp gecastet.

Beachten Sie außerdem, dass bei verketteten Operationen die Konvertierung je Operator ausgeführt wird, nicht für die gesamte Kette. Auch die Bedeutung dieses Details wird am Beispiel schnell klar: das Ergebnis von `(2 / 5) + 0.3` ist 0,3, da die Division ganzzahlig ausgeführt wird und das Ergebnis 0 hat.

Es ist nicht notwendig, die Feinheiten der Konvertierung auswendig zu kennen. Sie sollten aber immer im Hinterkopf behalten, wann sie stattfinden und wann nicht, denn auch hieraus entstehen Fehler, die nur schwer zu finden sind. Bedenken Sie

außerdem immer: **Der Typ der Variablen, der das Ergebnis zugewiesen wird, hat keinen Einfluss auf den Typ des Rechenergebnisses.** Um im Divisionsergebnis Dezimalstellen zu erhalten, reicht es nicht aus, es einer `double`-Variablen zuzuweisen.

Kurzschreibweisen

Es gibt eine Kurzschreibweise für den häufigen Fall, dass eine Berechnung mit einer Variablen durchgeführt und das Ergebnis wieder in derselben Variablen gespeichert wird. Ein häufiger Fall ist, Schleifendurchläufe zu zählen: `durchlauf = durchlauf + 1`.

Erinnern Sie sich an den Unterschied zwischen dem mathematischen Gleichheitszeichen und dem Zuweisungsoperator in Java: In Java ist diese Berechnung sinnvoll, es wird zuerst die rechte Seite der Zuweisung berechnet, das Ergebnis wird in der Variablen links gespeichert.

Da dieser Fall so häufig ist (und Programmierer notorisch schreibfaul sind) gibt es dafür die Kurzschreibweise `durchlauf += 1`. Beide Anweisungen haben dieselbe Bedeutung. Analoge Kurzschreibweisen existieren auch für die anderen Rechenoperatoren, aber nicht alle kommen häufig zum Einsatz. Ich warte bis heute auf eine Gelegenheit, den `%=`-Operator einzusetzen ...

Für die besonders häufigen Fälle Addition und Subtraktion von 1 gibt es mit den Operatoren `++` und `--` eine noch kürzere Schreibweise: `i++` hat dieselbe Bedeutung wie `i = i + 1`, `i--` wie `i = i - 1`.

Beide Operatoren haben gegenüber ihren Langschreibweisen eine weitere Einsatzmöglichkeit, sie können auch innerhalb einer anderen Berechnung als Seiteneffekt verwendet werden. So führt zum Beispiel die Anweisung `int j = i++ * 2` dazu, dass der alte Wert von `i` verdoppelt und das Ergebnis der Variablen `j` zugewiesen wird. Es wird aber auch der Wert von `i` um eins inkrementiert. Wie gezeigt findet das Inkrementieren von `i` nach der Berechnung statt. Der `++`-Operator kann aber auch vor der Variablen stehen, in diesem Fall wird zuerst inkrementiert und dann mit dem neuen Wert weitergerechnet. Vergleichen Sie dazu die beiden Codefragmente:

```
int i = 5;
int j = i++ * 2;
```

Listing 2.11 Inkrementieren mit Suffix

```
int i = 5;
int j = ++i * 2;
```

Listing 2.12 Inkrementieren mit Präfix

Im ersten Fragment wird zuerst der Wert für `j` berechnet, anschließend wird `i` um 1 erhöht. Die Werte der Variablen sind j = 10 und i = 6. Im zweiten Fragment wird zuerst

`i` inkrementiert und anschließend die Multiplikation ausgeführt, das Resultat ist j = 12 und i = 6. Das Gleiche gilt auch für den Operator `--`.

2.2.3 Bit-Operatoren

Für ganzzahlige Typen existieren weitere Operatoren, die auf der Binärdarstellung der Zahl operieren. Die Effekte dieser Operatoren sind gut zu veranschaulichen, wenn Sie die Zahlenwerte in Binärschreibweise angeben. Alle diese Operatoren führen dieselbe Typumwandlung durch, die auch bei den Rechenoperationen durchgeführt werden. Konkret bedeutet das, dass die Ergebnisse immer vom Typ `int` oder `long` sind, auch wenn die Eingabetypen `short` oder `byte` waren.

Bitweise Negation

Der Operator `~` schaltet jedes Bit einer Zahl um, setzt also eine 1 für jede 0 und eine 0 für jede 1. Aus `~0b101` wird so die etwas unhandliche Zahl `0b11111111111111111111111111111010`. Würden Sie für das Literal den Typ `long` erzwingen, also `~0b101L` angeben, so wäre das 64-bittige Ergebnis noch unhandlicher.

Bitweise Verknüpfungen

Diese Gruppe von Operatoren verknüpft zwei Ganzzahlen Bit für Bit mit den logischen Verknüpfungen AND, OR und XOR. Das Ergebnis ist eine Zahl, deren n-tes Bit das Ergebnis der Verknüpfung der n-ten Bits der Operanden ist.

Operator	Verknüpfung	Bedeutung	Beispiel
`&`	AND	Ein Bit im Ergebnis ist gesetzt, wenn das entsprechende Bit in beiden Operanden gesetzt ist.	`0b0011 & 0b0101 = 0b0001`
`|`	OR	Ein Bit im Ergebnis ist gesetzt, wenn das entsprechende Bit in einem oder beiden Operanden gesetzt ist.	`0b0011 | 0b0101 = 0b0111`
`^`	XOR	Ein Bit im Ergebnis ist gesetzt, wenn das entsprechende Bit in genau einem der Operanden gesetzt ist.	`0b0011 ^ 0b0101 =0b0110`

Tabelle 2.4 Übersicht über die bitweisen Verknüpfungen

Shift-Operatoren

Die Shift-Operatoren verschieben die Bits innerhalb einer Zahl nach links (`<<`) oder rechts (`>>`), machen also aus `0b0110` entweder `0b1100` oder `0b0011`. Dabei wird jeweils um so viele Bits verschoben, wie der zweite Operand angibt.

`0b00011010 << 2 = 0b01101000` *(Verschiebung um 2 Bit nach links)*
`0b00011010 >> 3 = 0b00000011` *(Verschiebung um 3 Bit nach rechts)*

Bits, die dabei herausgeschoben werden, gehen verloren. Beim *shift left* wird an der rechten Seite immer mit 0 aufgefüllt. Beim *shift right* dagegen wird links mit dem Wert des Vorzeichen-Bits aufgefüllt. Der Grund dafür liegt in der mathematischen Bedeutung der Operation, denn mathematisch gesehen entspricht jedes Verschieben nach links einer Multiplikation mit zwei, Verschieben nach rechts einer Division durch zwei. Damit das auch mit negativen Zahlen funktioniert, ist es notwendig, negative Zahlen von links mit 1 aufzufüllen, positive aber mit 0. Das liegt an der Darstellung negativer Zahlen im Zweierkomplement.

Sollten Sie das Vorzeichen ignorieren und auch von links immer mit 0 auffüllen wollen, so gibt es dafür den Operator `>>>`, der genau das tut.

Auch für die Shift-Operatoren gibt es übrigens eine Kurzschreibweise: `eineVariable <<= 2` entspricht `eineVariable = eineVariable << 2`.

[!]

Achtung: Zweierkomplement und negative Zahlen

Ganzzahlen werden im Zweierkomplement dargestellt. Das höchste Bit jeder Zahl dient dabei als Vorzeichen-Bit; steht dort eine 0, handelt es sich um eine positive, bei einer 1 um eine negative Zahl. Es reicht aber nicht, das Vorzeichen-Bit umzustellen, um eine Zahl in ihr negatives Gegenstück umzustellen. Um eine positive Zahl ins Negative umzukehren, lautet der Algorithmus:

1. alle Bits der Zahl negieren, also 0 in 1 und 1 in 0 wandeln
2. 1 hinzuaddieren

In 8 Bit wird so aus 01111111 = 127 die negative Zahl 10000001 = –127.

Wenn Sie negative Zahlen in Binärschreibweise angeben, dann müssen Sie diese aber nicht selbst ins Zweierkomplement umrechnen, das Minuszeichen funktioniert auch. Möchten oder müssen Sie es doch tun, müssen Sie die volle Bit-Länge des Datentyps angeben, also zum Beispiel 32 Bit für einen `int`-Wert:

`-2 = -0b10 = 0b11111111111111111111111111111110`

2.2.4 Übung: Ausdrücke und Datentypen

Welchen Datentyp und Wert haben die folgenden Ausdrücke? Lösungen zu allen Übungen finden Sie im Anhang.

1. `7`
2. `7.0`
3. `7L`

4. `5 + 9`
5. `5 + 9.0`
6. `5 + 9.0f`
7. `5d + 9.0f`
8. `(byte)5 + (short) 9`
9. `17 << 2`
10. `7L * 2`
11. `5 / 2`
12. `5 / 2.0`
13. `int i = 0; i--;`

2.2.5 Character-Variablen

Ein Character (Datentyp `char`) ist ein einzelnes Unicode-Zeichen, sei es ein Buchstabe, eine Ziffer, ein Satzzeichen oder Sonstiges. Eine `char`-Variable belegt 16 Bit und ist vorzeichenlos, das heißt, sie kann 65.535 mögliche Werte annehmen. `char` verwendet also UTF-16, so wie Java es intern insgesamt tut.

Unicode

Unicode ist eine Sammlung von *Character Encodings*, also von Übersetzungstabellen, um Buchstaben, Ziffern usw. in eine Reihe von Nullen und Einsen umzuwandeln, die ein Computer verarbeiten kann. Es gibt weltweit eine Vielzahl von Schriftsystemen, Sonderzeichen, Zeichenkombination sowie wissenschaftlichen und anderen Symbolen, die Unicode versucht, alle in einer Zeichentabelle zu vereinen. Unicode umfasst alle in der Welt gebräuchlichen Zeichen, von unserem lateinischen über das kyrillische und arabische Alphabet bis hin zu chinesischen und japanischen Zeichen und weit darüber hinaus.

Dadurch, dass sich alle in der Welt gebräuchlichen Zeichen darin finden, hat Unicode es endlich geschafft, einen weltweiten Standard für Zeichencodierung zu etablieren. Datenaustausch wurde dadurch immens vereinfacht. Es gibt verschiedene Ausprägungen von Unicode, die wichtigsten sind UTF-8, UTF-16 und UTF-32. Die Zahl gibt jeweils an, wie viel Bit zur Codierung eines Zeichens benötigt werden.

Um im Code `char`-Literale anzugeben, setzen Sie das Zeichen in **einfache** Anführungszeichen:

```
char beispiel = 'A';
```

Sie können auch Zeichen angeben, die Sie auf Ihrem Computer nicht eingeben oder nicht darstellen können, indem Sie direkt den Unicode des Zeichens mit dem Präfix

\u angeben. Den Code des Zeichens können Sie den Zeichentabellen des Unicode Consortiums auf *http://unicode.org/* entnehmen.

```
//der arabische Buchstabe Alef
char arabischesBeispiel = '\u0627';
```

Ein char kann nur genau ein Unicode-Zeichen aufnehmen. Das heißt, dass nicht alles, was Sie im Alltag als ein Zeichen ansehen würden, auch in einen char passt. Betroffen sind zwei Gruppen von Zeichen: kombinierte Zeichen, zum Beispiel Zeichen mit Diakritika wie »ž«, und Zeichen, die in UTF-16 als mehrere Zeichen umschrieben werden müssen. Dieser zweite Fall ist leider notwendig, da Unicode mehr als 65.535 Zeichen umfasst. Im Alltag kommen Sie damit allerdings nur selten in Berührung.

char-Werte sind zwar Zeichenvariablen, aber sie lassen sich auch begrenzt als Zahlenwerte behandeln. Sie können zwei char-Werte oder auch einen char und einen int addieren oder subtrahieren. Das klingt zunächst wenig nützlich, es gibt aber durchaus häufige Anwendungen, zum Beispiel eine Schleife über alle Kleinbuchstaben.

```
for (char c = 'a'; c <= 'z'; c++){
    System.out.println(c);
}
```

Listing 2.13 Eine »for«-Schleife über alle Kleinbuchstaben

2.2.6 Boolesche Variablen

Der letzte primitive Datentyp ist auch gleichzeitig der eingeschränkteste. boolean-Variablen enthalten Wahrheitswerte, es gibt also nur true (wahr) und false (falsch).

Speicherbedarf eines Boolean

Obwohl klar ist, dass ein boolean-Wert genau 1 Bit darstellt, beträgt der wirklich belegte Speicher auf den meisten JVMs stolze 4 Byte.

```
boolean wahr = true;
boolean falsch = false;
```

Listing 2.14 Die zwei möglichen »boolean«-Werte

Es gibt eine Reihe von Operatoren speziell für boolean-Werte, die uns in Kapitel 3, »Entscheidungen«, ausführlich beschäftigen werden.

2.2.7 Vergleichsoperatoren

Es gibt eine weitere Gruppe von Operatoren, die Sie auf Primitivtypen anwenden können: die Vergleichsoperatoren (siehe Tabelle 2.5).

Operator	Bedeutung	Zieldatentypen
==	gleich	alle Datentypen
!=	ungleich	alle Datentypen
<	kleiner als	byte, short, int, long, char, float, double
<=	kleiner als oder gleich	byte, short, int, long, char, float, double
>=	größer als oder gleich	byte, short, int, long, char, float, double
>	größer als	byte, short, int, long, char, float, double

Tabelle 2.5 Vergleichsoperatoren in der Übersicht

Vergleiche haben immer ein boolesches Ergebnis, das heißt, ein Vergleich ist immer wahr oder falsch. Verwendet werden Sie hauptsächlich im Zusammenhang mit Entscheidungen, die Sie in Kapitel 3 kennenlernen werden. Ein kurzes, vorgezogenes Beispiel macht diese Verwendung von Vergleichen klarer:

```
int zahl = …;
if (zahl < 0){
    //dieser Block wird nur ausgeführt, wenn zahl negativ ist.
}
```

Listing 2.15 Ein »if«-Statement mit Vergleich

Sie können das Ergebnis eines Vergleichs aber auch einer boolean-Variablen zuweisen, genau wie Sie das Ergebnis einer anderen Operation einer passenden Variablen zuweisen:

```
boolean positiv = zahl > 0;
```

Die Vergleichsoperatoren halten keine großen Überraschungen bereit. Die einzige Stolperfalle betrifft die Typen float und double. Für diese ist der Test auf Gleichheit mit == zwar möglich, führt aber häufig nicht zum gewünschten Ergebnis. Betrachten Sie dazu das folgende Beispiel:

```
if (1.000001 - 0.000001 == 1){
    System.out.println("Es funktioniert");
}
```

Listing 2.16 Vergleich von float und double: So geht es nicht.

Obwohl ohne Taschenrechner zu erkennen ist, dass die Differenz 1 ist, scheitert der Vergleich. Der Rumpf des `if`-Statements wird nicht ausgeführt. Das liegt daran, dass die Fließkommatypen nicht jeden Wert exakt darstellen können, manche Werte werden nur angenähert. Das Ergebnis der gezeigten Rechnung ist dadurch nicht 1, sondern 0,9999999999999999. Aus diesem Grund sollten Sie `float` und `double` nie auf Gleichheit prüfen, sondern immer einen Deltavergleich durchführen:

```
if (Math.abs((1.000001 - 0.000001) - 1) < 0.000000001){
    System.out.println("Es funktioniert");
}
```

Listing 2.17 Vergleich von »float« und »double«: So wird's gemacht.

Sie prüfen so, ob das Ergebnis mit einer für Sie akzeptablen Genauigkeit dem Vergleichswert entspricht.

2.3 Objekttypen

Alle Typen außer den acht oben beschriebenen sind Objekte. Objekte sind, wie bereits in Kapitel 1, »Einführung«, beschrieben, Datenstrukturen, die zusammengehörige Daten und Methoden zusammenfassen, die auf diesen Daten arbeiten. Wie Sie bereits in den Beispielen in Kapitel 1 gesehen haben, werden Objekte mit dem `new`-Operator aus einer Klasse erzeugt. (Die einzige Ausnahme davon sind `String`-Literale und Wrapper-Typen, die durch Autoboxing erzeugt wurden. Zu beidem folgt unten mehr.)

Eine Variable eines Objekttyps deklarieren Sie auf dieselbe Art wie eine primitive Variable: Variablentyp, gefolgt vom Namen.

```
Object meinErstesObjekt = new Object();
```

Listing 2.18 Deklaration und Zuweisung einer Objektvariablen

Trotz der gleichen Syntax unterscheiden sich Objektvariablen aber grundsätzlich von primitiven Variablen: Primitive Variablen speichern Werte, Objektvariablen speichern Referenzen.

2.3.1 Werte und Referenzen

Variablen sind, wie bereits gesagt, benannte Speicherstellen. Für primitive Typen bedeutet das, dass an dieser Speicherstelle der aktuelle Wert der Variablen gespeichert wird. Bei einer Objektvariablen steht an dieser Speicherstelle aber nicht das Objekt, sondern nur eine weitere Speicheradresse, an der sich das Objekt befindet. Man spricht von einer *Referenz* oder einem *Zeiger* auf das Objekt.

Auf den ersten Blick erscheint dieser Unterschied für Sie als Anwendungsentwickler nicht relevant, aber daraus folgt der wichtige Unterschied zwischen den beiden Arten von Variablen. Betrachten Sie dazu die folgenden Beispiele.

```
int zahl1 = 5;
int zahl2 = zahl1;
```

Listing 2.19 Zuweisung von Primitivtypen

```
BeispielObjekt objekt1 = new BeispielObjekt();
BeispielObjekt objekt2 = objekt1;
```

Listing 2.20 Zuweisung von Objekten

In beiden Fällen wird zuerst einer Variablen ein Wert zugewiesen und dann der Wert dieser Variablen einer weiteren Variablen zugewiesen. Die Variable `zahl2` enthält anschließend eine Kopie des Wertes von `zahl1`, also 5. `objekt2` enthält eine Kopie des in `objekt1` gespeicherten *Zeigers*, es wird aber keine Kopie des *Objekts* erstellt.

Dieser Unterschied wird dadurch wichtig, dass Objekte Daten enthalten, die von außen verändert werden können. Nehmen Sie an, die Klasse `BeispielObjekt` deklariert ein Feld namens `zahl` vom Typ `int`. Sie können auf dieses Feld mit dem Punktoperator zugreifen und es genauso behandeln wie jede andere Variable. Zum Beispiel können Sie einen Wert zuweisen:

```
objekt2.zahl = 21;
```

Was ist nun das Ergebnis, wenn Sie den Wert von `objekt1.zahl` auslesen? Bedenken Sie, dass keine Kopie des Objekts erstellt wurde, es existiert nur ein Exemplar von `BeispielObjekt` im Speicher – man spricht von einer *Instanz* –, und beide Variablen zeigen darauf. Der Wert von `objekt1.zahl` ist demnach 21.

2.3.2 Der Wert »null«

Die Funktionsweise von Objektvariablen als Referenz auf ein Objekt erklärt auch den Sinn des speziellen Wertes `null`. Er bedeutet, dass eine Variable auf kein Objekt zeigt. Mit einem `null`-Wert in einer Variablen können Sie nicht viel anfangen. Jeder Versuch, auf Felder oder Methoden zuzugreifen, führt zum wahrscheinlich häufigsten

Laufzeitfehler beim Arbeiten mit Java: der `NullPointerException`. Der Compiler kann Sie vor diesem Fehler nicht schützen, er führt keine Analyse auf `null`-Werte durch.

```
Object obj = null;
obj.toString();
```

Listing 2.21 Der schnellste Weg zur »NullPointerException«

Dieses Codefragment wird problemlos kompiliert. Der Compiler prüft nur, ob der deklarierte Typ der Variablen `obj` die Methode `toString` enthält. Das tut er, `toString` ist eine der Methoden, die an jedem Objekt existieren. Erst zur Laufzeit kommt es zur `NullPointerException`.

Ist eine Variable `null`, so bedeutet dass ganz allgemein, dass sie keinen Wert hat, aber man erfährt nicht, warum das so ist. Es kann bedeuten, dass die Variable schlicht noch nicht initialisiert wurde oder dass eine Operation kein Ergebnis hat, zum Beispiel weil bei einer Suche keine passenden Ergebnisse gefunden wurden.

2.3.3 Vergleichsoperatoren

Auch für Objekttypen existieren die Vergleichsoperatoren `==` und `!=`. Allerdings bedeuten Gleichheit und Ungleichheit für Objekte nicht, dass sie wertgleich sind, sondern dass die beiden Zeiger dasselbe Objekt referenzieren. Auch wenn zwei Objekte vom selben Typ sind und alle ihre Felder wertgleich, so sind die Objekte doch nicht gleich im Sinne des Gleichheitsoperators, sondern nur in diesem Fall:

```
Object obj1 = new Object();
Object obj2 = new Object();
if (obj1 == obj2){
    //dieser Block wird nicht ausgeführt
}
```

Listing 2.22 Wertgleich, aber nicht dasselbe Objekt

```
Object obj1 = new Object();
Object obj2 = obj1;
if (obj1 == obj2){
    …
}
```

Listing 2.23 Nur so sind zwei Objekte gleich.

Die Prüfung auf Wertgleichheit zweier Objekte wird in Java nicht mit einem Operator umgesetzt, sondern mit der `equals`-Methode, die Sie in Kapitel 5, »Klassen und Objekte«, kennenlernen werden.

2.3.4 Allgemeine und spezielle Typen

Bisher haben alle Beispiele nahegelegt, dass eine Variable nur auf Objekte genau des Typs verweisen kann, mit dem sie deklariert wurde. Das ist nicht die ganze Wahrheit. Es gibt unter Klassen Vererbungsbeziehungen, die uns beim Thema Objektorientierung (siehe Kapitel 6) ausführlich beschäftigen werden. Eine Klasse, die von einer anderen erbt, enthält alle Felder und Methoden dieser Klasse, kann aber noch eigene hinzufügen. Man sagt deshalb auch, dass sie die Klasse, von der sie erbt, *erweitert*.

Ein klassisches Beispiel, um Vererbung zu veranschaulichen, sind Fahrzeuge. Ein Auto ist eine spezielle Art von Fahrzeug, wenn Sie diesen Sachverhalt objektorientiert darstellen wollen, dann lassen Sie die Klasse `Auto` von der Klasse `Fahrzeug` erben. Von Auto kann es wiederum eine weitere Spezialisierung geben, zum Beispiel den Sportwagen. Die Klasse `Sportwagen` erbt von `Auto` und dadurch indirekt auch von `Fahrzeug`. Im Java-Code ermöglicht dieser Zusammenhang folgendes Codefragment:

```
Sportwagen meinSportwagen = new Sportwagen();
Auto auchMeins = meinSportwagen;
Fahrzeug immerNochMeins = meinSportwagen;
```

Listing 2.24 Objektvariablen und Vererbung

Diese Zuweisungen sind möglich, obwohl die Variablen verschiedene Typen haben, weil diese Typen in einer Vererbungsbeziehung stehen. Beachten Sie aber, dass dies nur in eine Richtung funktioniert, der folgende Code ist falsch:

```
Auto meinAuto = new Auto();
Sportwagen meinSchnellesAuto = meinAuto;
```

Listing 2.25 Diese Zuweisung funktioniert nicht.

Warum die Zuweisung in eine Richtung funktioniert und in die andere nicht, ist mit Alltagswissen leicht zu erklären: Jeder Sportwagen ist auch ein Auto, aber nicht jedes Auto ist ein Sportwagen. Deshalb können Sie von `Sportwagen` an `Auto` zuweisen, nicht aber von `Auto` an `Sportwagen`, denn eine Variable vom Typ `Auto` kann zwar auf ein Objekt vom Typ `Sportwagen` verweisen, muss es aber nicht.

Im JDK kommen zwar nur wenige Autos vor, aber derselbe Zusammenhang besteht auch zwischen Klassen aus der Klassenbibliothek. Ein Beispiel haben Sie in Kapitel 1, »Einführung«, schon gesehen:

```
private Map wordCounts = new HashMap();
```

Diese Zeile können Sie sich jetzt sofort erklären: `HashMap` erbt von `Map`, deshalb ist diese Zuweisung möglich (streng genommen implementiert `HashMap` das Interface `Map`, aber zu diesem Zeitpunkt ist der Unterschied noch nicht relevant).

Typumwandlung

Was kann man aber tun, wenn man doch ein `Auto` einer `Sportwagen`-Variablen zuweisen möchte und sich völlig sicher ist, dass es auch ein `Sportwagen` ist?

Dafür gibt es auch bei Objekten die Möglichkeit zu casten, indem Sie in Klammern den Zieltyp angeben.

```
Auto meinAuto = new Sportwagen();
Sportwagen schnellesAuto = (Sportwagen) meinAuto;
```

Listing 2.26 Casten von Objekten

Im Gegensatz zum Casten bei primitiven Typen wird das Objekt aber hierdurch nicht in einen anderen Typ konvertiert. Dazu gibt es keine Möglichkeit. Damit der Cast funktioniert, muss das Objekt schon vom richtigen Typ sein.

Aber wozu ist ein Cast bei Objekttypen dann überhaupt gut? Wie oben bereits erwähnt, kann eine Klasse von einer anderen erben und sie um eigene Felder und Methoden erweitern. Wenn das Objekt aber in einer Variablen vom Typ der Oberklasse gespeichert ist, kann auf diese Erweiterungen nicht zugegriffen werden, denn der Compiler lässt nur Zugriffe zu, die zum *deklarierten* Typ der Variablen passen. Auch hierzu wieder ein Beispiel. Nehmen Sie an, dass `Fahrzeug` die Methode `fahre()` implementiert. Durch die Vererbung kennt auch `Sportwagen` diese Methode. Zusätzlich kennt `Sportwagen`, und nur `Sportwagen`, auch die Methode `fahreSchnell()`.

```
Auto meinAuto = new Sportwagen();
meinAuto.fahre(); //Dieser Aufruf funktioniert
meinAuto.fahreSchnell(); //Kompiliert nicht
((Sportwagen)meinAuto).fahreSchnell(); //funktioniert
```

Listing 2.27 Wozu braucht man Casts bei Objekten?

In diesem Fall funktioniert der Cast, da `meinAuto` wirklich einen Sportwagen referenziert. Würde die Variable auf ein einfaches `Auto` oder auf eine andere Subklasse von `Auto` verweisen, dann käme es zur Laufzeit zu einer `ClassCastException`.

2.3.5 Strings – primitive Objekte

Unter den Objekten genießen Strings, also Zeichenketten, eine Sonderstellung. Als einziger Objekttyp gibt es für sie eine Literalschreibweise und einen eigenen Operator, die Konkatenation mit +. String-Literale haben Sie in vielen Beispielen bereits gesehen, die Zeichenkette wird in doppelten Anführungszeichen angegeben. Aus dem String-Literal wird ein Objekt vom Typ `String` erzeugt.

Der Konkatenationsoperator erzeugt aus zwei Strings eine neuen, indem er beide Strings hintereinanderschreibt. Ausführlich werden wir uns mit Strings in Abschnitt 8.2 beschäftigen. Merken Sie sich aber schon jetzt, dass Strings Objekte sind, denn die Warnung über Objektvergleiche mit == gilt auch für Strings.

2.4 Objekt-Wrapper zu Primitiven

Die Trennung zwischen Primitivtypen und Objekttypen ist auf den ersten Blick total. Auf der einen Seite stehen die Primitiven, auf der anderen Seite die Objekte. Das ist zwar richtig, aber es gibt eine Brücke zwischen den beiden Welten, die sogenannten *Wrapper-Klassen*. Schon seit Java 1.0 (1.1 für `byte` und `short`) gibt es zu jedem Primitivtyp eine Klasse, die diesen Typ in ein Objekt »verpacken« kann, um damit objektorientiert arbeiten zu können (siehe Tabelle 2.6).

Primitivtyp	Klasse
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

Tabelle 2.6 Primitivtypen und ihre Wrapper-Klassen

2.4.1 Warum?

Dieses Doppelleben der Primitivtypen ist nicht so sinnlos, wie es zunächst den Anschein hat. Sowohl Primitive als auch Objekte haben ihre Vorteile. Die Primitiven sind speichersparender und performanter als Objekte, für die am häufigsten genutzten Datentypen nicht unwichtig. Andererseits können Objekte nützliche Methoden enthalten und Klassen wichtige Konstanten definieren. Zum Beispiel enthalten die Wrapper zu den numerischen Typen jeweils die Konstanten `MIN_VALUE` und `MAX_VALUE`, die den kleinsten bzw. größten Wert des Typs enthalten. Auch finden Sie in den

Wrapper-Klassen der Zahlentypen die sehr nützlichen `parse*`-Methoden (`Integer.parseInt()`, `Short.parseShort()`, `Double.parseDouble()`, ...), die einen `String` in den jeweiligen Zahlentyp umwandeln, soweit möglich.

Die Spaltung in Primitive und Objekte war in den Anfängen von Java der Weg, das Beste beider Welten zu erhalten. Um die Vorteile beider Seiten nutzen zu können, ist natürlich ein Weg notwendig, zwischen Primitiven und ihren Wrappern zu konvertieren.

2.4.2 Explizite Konvertierung

Alle Wrapper-Klassen enthalten Methoden, um zwischen Primitiven und Wrappern zu konvertieren. Vom Primitiven zum Objekt kommen Sie immer mit der statischen Methode `valueOf` der jeweiligen Wrapper-Klasse (Sie erinnern sich, statische Methoden sind solche, die direkt an einer Klasse aufgerufen werden können). `Integer.valueOf(21)` erzeugt ein `Integer`-Objekt, das den Wert 21 enthält. `Boolean.valueOf(false)` tut dasselbe mit dem Wert `false` in einem `Boolean`-Objekt usw.

In die andere Richtung gibt es am Objekt – diese Methoden sind nicht statisch – jeweils eine Methode `*Value()` (zum Beispiel `intValue()`, `charValue()`, ...), die wieder einen Primitivwert daraus macht.

[!]

Konvertieren von Primitiven in Wrapper-Objekte

Es gibt an den Wrapper-Klassen auch jeweils einen Konstruktor, um aus einem Primitiven ein Objekt zu erzeugen. Verwenden Sie diesen nicht. Er funktioniert zwar, aber die `valueOf`-Methoden können Objekte wiederverwenden. Wenn Sie mehrmals `Integer.valueOf(21)` rufen, dann bekommen Sie möglicherweise dasselbe Objekt zurück, so wird Speicher gespart. Mit dem Konstruktor ist das nicht möglich, er *muss* immer ein neues Objekt erzeugen. Da Instanzen der Wrapper-Klassen unveränderlich sind, können keine Probleme dadurch auftreten, dass sie wiederverwendet werden.

2.4.3 Implizite Konvertierung

Seit Java 5 müssen Sie die Konvertierung in vielen Fällen nicht mehr selbst durchführen. Seit dieser Version gibt es *Autoboxing* bzw. *Autounboxing*, einen Mechanismus, der die Umwandlung nach Bedarf durchführt. Seitdem können Sie zum Beispiel Zuweisungen wie `Integer zahlenObjekt = 5` ausführen, aus dem `int`-Wert `5` wird automatisch ein `Integer`-Objekt erzeugt. Auch andersherum funktioniert es:

```
Integer zahl = 5;
int summe = zahl + 1;
```

Listing 2.28 Autoboxing und -unboxing

In der zweiten Zeile des Codefragments wird `zahl` automatisch in einen `int` umgewandelt, denn `Integer`-Objekte können nicht mit `+` addiert werden. Allerdings funktioniert die Umwandlung nicht in allen Situationen, `13.toString()` ist beispielsweise nach wie vor nicht möglich.

Implizite Konvertierung und null-Werte

Besondere Vorsicht ist nötig, wenn Sie mit Wrapper-Objekten und `null`-Werten arbeiten. Wie bei allen anderen Objekttypen ist `null` auch für die Wrapper ein gültiger Wert. Primitive können aber nie `null` sein. Das ist sogar ein weiterer Grund, die Wrapper zu benutzen: `null`-Werte sind wertvoll, sie können zum Beispiel aussagen, dass eine Berechnung noch nicht ausgeführt wurde. Mit Primitiven lässt sich das nicht ausdrücken. Da es aber keine primitive Entsprechung zu `null` gibt, schlägt der Versuch zu »unboxen« mit einer `NullPointerException` fehl:

```
Integer zahl = null;
Integer zahl2 = zahl +1;
```

Listing 2.29 NullPointerException beim Unboxing

Dieser Code kompiliert problemlos, der Fehler tritt erst zur Laufzeit auf. Als Entwickler müssen Sie entscheiden, wie Sie damit umgehen. Wenn Ihre Anwendung `null` an dieser Stelle nicht erlaubt, dann können Sie auf die Prüfung verzichten und die `NullPointerException` auftreten lassen. Der Aufrufer der Methode erhält dann einen Fehler, den er behandeln kann. Sauberer ist es aber, die Prüfung selbst vorzunehmen, um eine bessere Fehlermeldung erzeugen zu können. Das funktioniert ganz ähnlich wie die Parameterprüfung, die Sie in den verschiedenen `main`-Methoden der bisherigen Beispiele gesehen haben.

```
public Integer verdopple(Integer in){
    if (in == null)
        throw new IllegalArgumentException("Null is not allowed");
    return in * 2
}
```

Listing 2.30 Prüfung auf null-Werte mit Werfen einer Exception

Ist `null` aber ein Wert, den Ihre Anwendung an dieser Stelle erlaubt, dann sollten Sie auch keine Fehler werfen. Stattdessen könnten Sie selbst wiederum `null` zurückgeben: Das Doppelte einer undefinierten Zahl ist undefiniert.

```
public Integer verdopple(Integer in){
    if (in == null)
        return null;
    return in * 2
}
```

Listing 2.31 Prüfung auf »null«-Werte mit »null« beantworten

Egal, wie Sie auf `null`-Werte reagieren, sollten Sie diese Reaktion im Javadoc der Methode festhalten.

2.5 Array-Typen

Alle gezeigten Typen, Objekte wie Primitive, haben gemeinsam, dass sie genau einen Wert des deklarierten Typs enthalten. Häufig wollen Sie aber auch mit mehreren zusammengehörigen Variablen desselben Typs arbeiten, zum Beispiel mit einer Liste von Namen oder einer Liste von Zahlenwerten, die die Koordinaten eines Punktes darstellen, oder mit einer Liste von `Point`-Objekten, die ein Polygon definieren.

In all diesen Fällen bietet es sich an, ein *Array* zu verwenden. Ausführlich werden wir uns mit Arrays (und deren Cousins, den Collections) in Kapitel 10 beschäftigen. Da es sich aber um einen grundlegenden Java-Datentyp handelt, sollen sie auch in diesem Kapitel nicht fehlen.

2.5.1 Deklaration eines Arrays

Arrays sind in Java immer an den eckigen Klammern `[]` zu erkennen. Sie werden sowohl in der Deklaration des Datentyps als auch zum Erzeugen eines neuen Arrays benötigt.

Sie können ein leeres Paar eckiger Klammern an jeden beliebigen Datentyp hängen, um ein Array dieses Typs zu bezeichnen, egal, ob es sich um einen primitiven Typ oder einen Objekttyp handelt. So ist `int[]` ein Array von Ganzzahlen, `String[]` ein Array von Zeichenketten und `MeineKlasse[]` ein Array von Objekten Ihrer selbst implementierten Klasse `MeineKlasse`. (Sie können die eckigen Klammern auch an den Variablennamen hängen, etwa so: `int meinArray[]`. Dies ist in Java aber extrem unüblich, tun Sie es nicht.)

Genau wie Objekte werden Arrays mit dem `new`-Operator erzeugt. Wenn Sie ein neues Array erzeugen, um es Ihrer Array-Variablen zuzuweisen, geben Sie in eckigen Klammern die Länge des Arrays an. Arrays haben eine feste, unveränderliche Länge zwischen 0 und 2.147.483.647. (Um genau zu sein, sind es, je nachdem, welche Laufzeitumgebung Sie verwenden, maximal 2.147.483.639 – 2.147.483.645 Einträge.

Auch wenn es an vielen Stellen so zu lesen ist, können Sie normalerweise nicht die volle Länge von 2.147.483.647 ausschöpfen.)

```
int[] zahlen = new int[100];
String[] namen = new String[512];
double[] koordinaten = new double[3];
```

Listing 2.32 Array-Deklarationen

Ein so deklariertes Array enthält an jeder Stelle den Default-Wert für den zugrunde liegenden Datentyp, also `null` für Objekte, 0 für Zahlenwerte und `false` für `boolean`.

Sie können aber auch für ein Array einen Startwert angeben, indem Sie in geschweiften Klammern die Werte auflisten. In diesem Fall müssen Sie keine Länge angeben.

```
double[] koordinaten = new double[]{1.5, 3.2, 1.0};
```

Listing 2.33 Array-Deklarationen mit Startwert

2.5.2 Zugriff auf ein Array

Auch beim Zugriff auf ein Array kommen die eckigen Klammern zum Einsatz. Sie hängen dem Namen der Array-Variablen in eckigen Klammern den Index des Elements an, auf das Sie zugreifen möchten. So können Sie Werte in einem Array genauso behandeln wie eine einfache Variable.

```
int[] zahlen = new int[]{1, 2, 3, 4, 5};
int ersteZahl = zahlen[0]; // den ersten Wert des Arrays lesen
zahlen[0] = 17; // den ersten Wert des Arrays schreiben
```

Listing 2.34 Zugriff auf Werte im Array

Beachten Sie, dass die erste Stelle des Arrays den Index 0 hat, nicht 1. Dadurch hat die letzte Stelle den Index Array-Länge –1 und **nicht** etwa Array-Länge. Wenn Sie versuchen, auf einen Index außerhalb des Arrays zuzugreifen, wird dies mit einer `IndexOutOfBoundsException` verhindert.

Die Länge des Arrays finden Sie in der Eigenschaft `length`. Sie können den Wert allerdings nur auslesen, nicht neu setzen.

```
int[] zahlen = new int[]{1, 2, 3, 4, 5};
System.out.println(zahlen.length + " Zahlen im Array");
```

Listing 2.35 Die Array-Länge auslesen

2.6 Zusammenfassung

In diesem Kapitel haben Sie gelernt, wie Sie Variablen deklarieren und mit ihnen arbeiten. Sie haben die acht primitiven Datentypen kennengelernt und gesehen, wie sie sich voneinander unterscheiden, wie Sie mit ihnen einfache Berechnungen anstellen und was der Unterschied zwischen den primitiven Typen und den Objekttypen ist. Im nächsten Kapitel werden Sie endlich beginnen, eigene Programme zu schreiben, und lernen, wie Sie in Ihrem Programm Entscheidungen treffen können.

Kapitel 8

Die Standardbibliothek

Mit den Grundlagen der Objektorientierung des letzten Kapitels haben Sie nun alle Voraussetzungen, um in Java zu programmieren. In diesem Kapitel werden Sie einige nützliche Helfer kennenlernen, die Ihnen dabei Arbeit abnehmen können. Javas Standardbibliothek enthält eine große Menge an Klassen, die Sie im Alltag immer wieder einsetzen werden, weil sie häufige Probleme elegant und einfach lösen.

Schon häufig wurde im Laufe des Buches die große, nützliche Standardbibliothek von Java erwähnt. Nun wird es Zeit, einige der nützlichsten Werkzeuge aus der Standardbibliothek kennenzulernen. Im weiteren Verlauf des Buches werden einige Teilbereiche der Standardbibliothek noch in eigenen Kapiteln behandelt, diese sind hier nicht weiter erwähnt.

8.1 Zahlen

Zahlen sind, auch wenn Ihre Programme keine mathematischen Funktionen umsetzen, der grundlegendste Baustein der Programmierung. Wenn Sie nur tief genug graben, dann lässt sich jede Operation, die Sie in einem Programm ausführen, auf Zahlen zurückführen. Entsprechend gibt es auch eine Reihe von Klassen in der Standardbibliothek, die diesem Umstand Rechnung tragen.

8.1.1 »Number« und die Zahlentypen

Die Wrapper-Klassen der Primitivtypen kennen Sie bereits aus Kapitel 2, »Variablen und Datentypen«. Auch wie Sie zwischen ihnen und den entsprechenden Primitiven konvertieren, haben Sie dort bereits gesehen. Aber die Wrapper-Klassen (`Byte`, `Short`, `Integer`, `Long`, `Float` und `Double`) bieten noch einige Kleinigkeiten mehr als nur die bereits erwähnten Methoden zur Konvertierung.

Zunächst einmal haben alle Wrapper-Klassen mit `Number` eine gemeinsame Oberklasse. Diese bietet zwar nicht viele Methoden, aber mit den vorhandenen Methoden `byteValue`, `doubleValue` usw. – analog für die restlichen Typen – können Sie zumindest jedes `Number`-Objekt in alle Arten von primitiven Zahlentypen übersetzen – unter

Runden und Abschneiden, versteht sich, auch diese Methoden können die Nachkommastellen eines `Double`-Objekts nicht in einem `int`-Wert unterbringen.

Auch die sehr nützlichen Konstanten `MIN_VALUE` und `MAX_VALUE` wurden bereits erwähnt. Sie enthalten in jeder der Wrapper-Klassen den minimalen und maximalen Wert, den eine Variable dieses Typs annehmen kann. Beachten Sie, dass `MIN_VALUE` bei den ganzzahligen Typen den kleinstmöglichen negativen Wert enthält, also den Wert, der am weitesten links der 0 liegt, bei den Fließkommatypen aber den positiven Wert, der am nächsten an der 0 liegt.

Die »TYPE«-Konstanten

In allen Wrapper-Klassen gibt es außerdem eine Konstante `TYPE`, die das Klassenobjekt des Primitivtyps enthält. Das klingt zunächst verwirrend, und das ist es auch ein wenig. Die Primitivtypen sind doch gerade keine Objekte, also haben sie auch keine Klasse. Warum gibt es dann dieses Klassenobjekt?

Sie sind eine notwendige Krücke für die Reflection-API. Unter anderem können Sie damit alle Methoden einer Klasse auflisten oder eine Methode an einem Objekt suchen, und dabei müssen Sie ausdrücken können, dass eine Methode einen primitiven Typ als Rückgabewert oder Parameter hat. Dazu dienen die `TYPE`-Konstanten der jeweiligen Wrapper-Klasse.

Außerdem gibt es an allen Wrapper-Klassen eine statische `parse`-Methode, um einen String in den jeweiligen Primitivtypen umzuwandeln. `Integer.parseInt` haben Sie bereits verwendet, analog dazu gibt es in den anderen Wrappern `Long.parseLong`, `Double.parseDouble` usw. Bei den `parse`-Methoden der Fließkommazahlen ist zu beachten, dass Trennzeichen nach dem englischen Standard verwendet werden: Ein Punkt trennt die Nachkommastellen ab. Möchten Sie flexibler sein und auch Zahlen in der deutschen Schreibweise akzeptieren, dann können Sie dazu die Klasse `NumberFormat` (siehe Unterabschnitt »Zahlen formatieren in ›MessageFormat‹« in Abschnitt 8.5.2) verwenden.

8.1.2 Mathematisches aus »java.lang.Math«

Eine Vielzahl an nützlichen mathematischen Funktionen finden Sie nicht als Methoden an den jeweiligen Klassen, sondern zusammengefasst in `java.lang.Math`, einer Klasse, die keinen anderen Zweck erfüllt, als statische Methoden für mathematische Operationen zu bündeln.

Nach der reinen Schule der Objektorientierung ist das nicht die schönste Lösung. Anstatt den Sinus eines Winkels mit `Math.sin(eineZahl)` zu berechnen, wäre es aus dieser Sicht besser, `eineZahl.sin()` rufen zu können. Zwei Gründe sprechen aber für die gewählte Lösung: Die Schnittstelle der Zahlentypen wird nicht mit einer großen

Zahl von Methoden aufgebläht, die in den meisten Programmen nicht benötigt werden, und Berechnungen müssen so nicht in mehreren Klassen implementiert werden, sondern nur genau einmal. Und so kommt es dann zu einer Klasse wie `Math`, die niemals instanziiert wird, sondern nur statische Hilfsmethoden bündelt.

Tabelle 8.1 bietet eine Übersicht über die wichtigsten Funktionen, die in `Math` realisiert sind. Manche dieser Funktionen sind für verschiedene Zahlentypen überladen, andere nehmen nur `double`-Parameter und verlassen sich darauf, dass andere Typen automatisch konvertiert werden können.

Funktion	Beschreibung
Exponentialfunktionen	
`Math.pow`	Berechnet für zwei Zahlen x und y die Potenz x^y.
`Math.sqrt`	Berechnet die Quadratwurzeleiner Zahl.
`Math.cbrt`	Berechnet die kubische Wurzel einer Zahl. Es gibt keine allgemeine Wurzelfunktion, wenn Sie die `n`-te Wurzel einer Zahl benötigen, so berechnen Sie diese nach der Formel $x^{1/n}$, also `Math.pow(x, 1/n)`. Achtung: Stellen Sie sicher, dass die Division ein Ergebnis vom Typ `double` hat, indem Sie `n` als double definieren oder casten.
`Math.log`	Berechnet den natürlichen Logarithmus einer Zahl, also den Logarithmus zur Basis e.
`Math.log10`	Berechnet den Logarithmus zur Basis 10.
Winkelfunktionen	
Alle Winkelfunktionen (außer den beiden letzten) erwarten Parameter in Bogenmaß und liefern auch Ergebnisse in Bogenmaß!	
`Math.sin`	Berechnet den Sinus eines Winkels.
`Math.cos`	Berechnet den Cosinus eines Winkels.
`Math.tan`	Berechnet den Tangens eines Winkels.
`Math.asin`	Berechnet den Arcus Sinus eines Winkels.
`Math.acos`	Berechnet den Arcus Cosinus eines Winkels.
`Math.atan`	Berechnet den Arcus Tangens eines Winkels.
`Math.toRadians`	Rechnet einen Winkel in Grad ins Bogenmaß um.

Tabelle 8.1 Die wichtigsten Rechenoperationen aus »java.lang.Math«

Funktion	Beschreibung
`Math.toDegrees`	Rechnet einen Winkel im Bogenmaß in Grad um.
Vergleichsfunktionen	
`Math.min`	Liefert die kleinere zweier Zahlen zurück.
`Math.max`	Liefert die größere zweier Zahlen zurück.
Rundungsfunktionen	
`Math.floor`	Gibt die nächstkleinere Ganzzahl zurück. Das Ergebnis ist echt kleiner, nicht im Betrag kleiner. Für negative Zahlen bedeutet das, dass das Ergebnis weiter von der 0 entfernt ist. Der Rückgabewert ist zwar vom Typ `double`, es wird aber immer eine ganze Zahl zurückgegeben.
`Math.ceil`	Gibt die nächstgrößere Ganzzahl zurück. Es gelten dieselben Anmerkungen wie bei `Math.floor`.
`Math.round`	Rundet einen `float`-Wert zum nächsten `int` oder einen `double`-Wert zum nächsten `long`.
Überlaufsichere Rechenoperationen	
`Math.addExact`	Addiert zwei `int`- oder `long`-Werte mit Überlaufsicherung: Wenn es zu einem Überlauf kommt, wird eine `ArithmeticException` geworfen. Es existiert keine analoge Methode für Fließkommazahlen, da es bei ihnen aufgrund der anderen internen Darstellung nicht zu Überläufen kommt.
`Math.subtractExact`	Subtrahiert zwei `int`- oder `long`-Werte mit Überlaufsicherung.
`Math.multiplyExact`	Multipliziert zwei `int`- oder `long`-Werte mit Überlaufsicherung.
Andere Funktionen	
`Math.abs`	Liefert den Betrag einer Zahl.
Konstanten	
`Math.E`	die eulersche Zahl e
`Math.PI`	die Kreiszahl Pi

Tabelle 8.1 Die wichtigsten Rechenoperationen aus »java.lang.Math« (Forts.)

8.1.3 Übung: Satz des Pythagoras

Verwenden Sie die Methoden von `Math`, um den Satz des Pythagoras ($a^2 + b^2 = c^2$) umzusetzen. Fragen Sie die Länge der beiden Katheten a und b vom Benutzer ab, und berechnen Sie die Länge der Hypotenuse c. Schreiben Sie Testfälle dazu. Die Lösung zu dieser Übung finden Sie im Anhang.

8.1.4 »BigInteger« und »BigDecimal«

Die größte Einschränkung der primitiven Zahlentypen ist ihr begrenzter Wertebereich. Selbst eine `long`- oder `double`-Variable hat Grenzen, welche Werte sie aufnehmen kann. Für genau diese Fälle enthält die Standardbibliothek zwei weitere Spezialisierungen von `Number`: `BigInteger` für unbegrenzt große Ganzzahlen und `BigDecimal` für unbegrenzt große (aber nicht unbegrenzt genaue!) Kommazahlen.

Beide Klassen bieten Konstruktoren, um neue Instanzen aus Strings zu erzeugen, sowie statische `valueOf`-Methoden, die Instanzen aus primitiven Zahlen erzeugen. Außerdem haben beide die Methoden `add`, `subtract`, `multiply` und `divide`. Diese ersetzen die Operatoren für die vier Grundrechenarten, da diese in Java nur für die primitiven Zahlentypen definiert sind. Um zwei `BigInteger` oder `BigDecimal` zu addieren, lautet der Code also `eineZahl.add(eineAndereZahl)`.

Diese Methoden verändern genau wie alle anderen Methoden beider Klassen niemals ein bestehendes Objekt. `BigInteger` und `BigDecimal` sind unveränderlich, alle Operationen lassen ihre Eingabeobjekte unverändert und geben ein neues Objekt mit dem berechneten Wert zurück.

Über diese grundlegenden Methoden hinaus haben `BigInteger` und `BigDecimal` unterschiedliche weitere Fähigkeiten.

BigInteger

`BigInteger` bietet Methoden zur Bit-Manipulation, analog zu den Operatoren für ganzzahlige Primitive: `and` (&&), `or` (||), `xor` (^), `not` (~), `shiftLeft` (<<) und `shiftRight` (>>). Darüber hinaus gibt es in `BigInteger` Methoden, die direkten Zugriff auf ein einzelnes Bit zulassen. `testBit` prüft, ob das Bit an der angegebenen Stelle gesetzt ist, also den Wert 1 hat. `setBit` und `clearBit` setzen das Bit an dieser Stelle auf 1 bzw. 0. `flipBit` setzt das angegebene Bit auf den entgegengesetzten Wert.

Sie können einen `BigInteger` also als Bit-Feld verwenden, genau wie einen herkömmlichen `int`-Wert, nur viel länger: Als Index für den Zugriff auf einzelne Bits wird ein `int` verwendet, Ihnen stehen also mehr als 2 Milliarden Stellen zur Verfügung.

BigDecimal

Ein `BigDecimal` wird als zwei Ganzzahlen gespeichert: der unskalierte Wert und die Skalierung. Ist die Skalierung eine positive Zahl, gibt sie an, nach wie vielen Stellen von rechts das Komma stehen soll. Zum Beispiel: Wert 314, Skalierung 2 stellt die Zahl 3,14 dar. Ist die Skalierung negativ, so wird sie als Exponent interpretiert in der Form Wert × $10^{\text{Skalierung}}$.

Es ist nicht wichtig, die genauen Interna von `BigDecimal` zu kennen, aber die Skalierung kann an viele Methoden als Parameter übergeben werden, man kann sie deshalb nicht völlig ignorieren. Zum Beispiel können Sie bei der Division angeben, was die Skalierung des Ergebnisses sein soll – wie viele Nachkommastellen es haben soll – und wie gerundet werden soll, falls das wirkliche Ergebnis mehr Stellen hat.

```
BigDecimal dividend = new BigDecimal("1");
BigDecimal divisor = new BigDecimal("32");
BigDecimal ergebnis = dividend.divide(divisor);
BigDecimal ergebnisRund =
 dividend.divide(divisor, 2, java.math.RoundingMode.DOWN);
```

Listing 8.1 Teilen mit und ohne Runden

Im Beispiel enthält die Variable `ergebnis` das exakte Teilungsergebnis 0,03125, die Variable `ergebnisRund` enthält den Wert 0,031. Wie gerundet werden soll, steuern Sie durch den übergebenen `RoundingMode`.

8.1.5 Übung: Fakultäten

Ein `long`-Wert klingt zunächst ausreichend für alle Berechnungen. Wo hat man es schon mit so großen Zahlen zu tun?

Schon mit einfachen Rechnungen können Sie aber sehr schnell in diese Bereiche vordringen. Beim Berechnen von Fakultäten kommt man sehr schnell zu sehr großen Zahlen. Die Fakultät einer Zahl n (n!) ist das Produkt der Zahl und aller kleineren, positiven Ganzzahlen. Zum Beispiel ist 6! = 6 × 5 × 4 × 3 × 2 × 1 = 720.

Schreiben Sie ein Programm, das Fakultäten berechnet, und zwar einmal mit `long` und einmal mit `BigInteger`. In beiden Fällen sollen Fakultäten der Zahlen von 1 bis 100 ausgegeben werden oder bis ein Überlauf auftritt, falls das vorher passiert. Die Lösung zu dieser Übung finden Sie im Anhang.

8.2 Strings

Wie Sie schon aus Kapitel 2, »Variablen und Datentypen«, wissen, ist `String` neben den Primitiven einer der grundlegenden Datentypen in Java. Strings sind Zeichenketten, sie enthalten Text. Dabei sind sie aber keine komplexen Datenstrukturen, ihr Inhalt wird in einem einfachen `char[]` gespeichert. Die `String`-Klasse bietet allerdings eine große Anzahl an Methoden, die auf diesem Array operieren.

Strings sind genau wie die Zahlentypen unveränderlich. Methoden, die den gespeicherten Text verändern, schreiben nie in das `char[]`, sondern geben immer ein neues Objekt zurück.

8.2.1 Unicode

Auch bei der Arbeit mit Zeichenketten darf man nicht vergessen, dass ein Computer nur mit Zahlen arbeitet. Um Zeichen und Zeichenketten im Speicher darstellen und manipulieren zu können, müssen sie also in Zahlen übersetzt werden. Diese Übersetzung erfolgt anhand von *Character Encodings*, einfachen Tabellen, die ein Zeichen einer Zahl zuordnen.

Traditionell umfassten Character Encodings 128 oder 256 Zeichen, so dass ein oder zwei Zeichen in einem Byte gespeichert werden konnten. Da das offensichtlich nicht ausreicht, um verschiedene Alphabete darzustellen, gab es viele verschiedene Encodings für verschiedene Alphabete und meist auch noch mehrere für ein Alphabet.

Java entgeht diesem unangenehmen Chaos von Character Encodings, indem es auf eine moderne Erfindung setzt: *Unicode*. Die Unicode Encodings stellen einen Versuch dar, alle bekannten Alphabete plus viele weitere Sonderzeichen aus Bereichen von Mathematik bis Astrologie in einer Tabelle zusammenzufassen. Es gibt mehrere verschiedene Encodings, die auf diese eine Tabelle zurückzuführen sind und sich darin unterscheiden, wie viele Byte mindestens für ein Zeichen verwendet werden. Die häufigsten Varianten sind UTF-8 mit mindestens 1 Byte (8 Bit) je Zeichen, UTF-16 mit mindestens 2 Byte (16 Bit) und UTF-32, das immer 4 Byte (32 Bit) je Zeichen verwendet. Beachten Sie dabei das Wort »mindestens«. In UTF-8 und UTF-16 werden viele Zeichen codiert, indem sie als mehrere Zeichen (richtiger als mehrere *Code Points*) gespeichert werden. Damit haben Sie glücklicherweise selten direkt zu tun, Sie arbeiten mit `String` oder `char`, und Java übernimmt die Details. Merken Sie sich nur, dass Sie nicht ohne Weiteres von der Anzahl der Zeichen in einem `String` auf seine Größe in Byte schließen können!

Java kann bei der Ein- und Ausgabe mit den verschiedenen Unicode Encodings und vielen weiteren umgehen, aber alle Strings liegen im Speicher in UTF-16 vor. Das bedeutet für Sie, dass Sie in einer Zeichenkette beliebige Mischungen der verschiedensten Schriftsysteme benutzen können. Zeichen, die Sie an Ihrem Computer nicht

eintippen können, können Sie auch in einem String durch die Notation \uxxxx ersetzen, wie bei einer char-Variablen. Dabei ersetzen Sie »xxxx« durch den Unicode des Zeichens.

```
String unicoded = "Dies ist eine Zeichenfolge aus dem Ogham-Alphabet: \u169B\
u1684\u1698\u169C";
```

Listing 8.2 Codierte Unicode-Zeichen in einem String

In der Ausgabe wird daraus »Dies ist eine Zeichenfolge aus dem Ogham-Alphabet: ᚛ᚁᚔᚘ᚜.«

[!]

Warnung zur Arbeit mit Unicode

Dass ein Zeichen im Unicode vorhanden ist, heißt noch nicht, dass Ihr Computer es auch darstellen kann. Dazu muss auch noch ein passender Zeichensatz installiert sein. Wenn Sie also mit arabischen, chinesischen und anderen Zeichen experimentieren, kann es vorkommen, dass Sie nur Fragezeichen oder leere Rechtecke sehen. Das ist dann kein Programmfehler, Ihr Computer kennt lediglich den Zeichensatz nicht.

8.2.2 String-Methoden

Die Klasse String enthält viele hilfreiche Methoden für fast alle Operationen, die Sie auf einer Zeichenkette durchführen wollen könnten. Wegen der zentralen Rolle der Klasse sollen im Folgenden die wichtigsten kurz aufgezählt werden. Es gibt über diese Methoden hinaus noch einige weitere, die mit *regulären Ausdrücken* arbeiten, mehr zu diesen erfahren Sie in Abschnitt 8.3.

length

Die length-Methode gibt die Länge des String zurück. Einige der anderen Methoden erwarten einen Index als Parameter, eine Stelle im String, an der sie operieren sollen. Bei all diesen Methoden ist wichtig, dass der Index zwischen 0 und length() -1 liegt, anderenfalls werfen sie eine StringIndexOutOfBoundsException.

charAt

Die Methode charAt erwartet einen Index als Parameter und gibt das Zeichen zurück, das an diesem Index steht. Sie kennen diese Methode bereits von den Beispielen, die alle Zeichen eines Strings in einer Schleife durchlaufen:


```
for (int i = 0; i < einString.length(); i++){
    char zeichen = einString.charAt(i);
}
```

Listing 8.3 Schleife über alle Zeichen eines Strings

»toUpperCase« und »toLowerCase«

Diese beiden Methoden konvertieren alle Zeichen des Strings in Großbuchstaben (toUpperCase) bzw. in Kleinbuchstaben (toLowerCase). Zeichen, für die es keine Groß- und Kleinschreibung gibt, zum Beispiel Ziffern, Sonderzeichen und Zeichen aus Schriften, die diese Unterscheidung nicht machen, bleiben unverändert.

Diese Operationen klingen trügerisch einfach, bergen aber unerwartete Tücken. Es gibt beispielsweise nicht immer eine 1:1-Zuordnung zwischen Klein- und Großbuchstaben. Nach einem Beispiel müssen Sie nicht erst in exotischen Schriftsystemen suchen, das deutsche »ß« gehört schon dazu: "ß".toUpperCase() ergibt "SS", aber "SS".toLowerCase() ergibt "ss".

Das ist zwar eher eine Kuriosität, aber die echten Probleme beginnen, wenn Sie verschiedene Lokalitäten betrachten, in denen das gleiche Zeichen anders behandelt wird. So wird im Deutschem aus einem »i« mit toUpperCase ein »I«. Im Türkischen aber ist der Großbuchstabe zu »i« das »İ« und der Kleinbuchstabe zu »I« das »ı«.

Es ist daher wichtig, zu wissen, nach den Regeln welcher Sprache diese Operationen ausgeführt werden. Dies wird in Java durch Locale-Objekte kontrolliert (siehe Kasten). Sie können toUpperCase und toLowerCase mit einem Locale-Parameter aufrufen, um festzulegen, welche Regeln angewendet werden sollen. Rufen Sie eine der Methoden ohne Parameter auf, dann wird das Default-Locale benutzt, das aus den Einstellungen Ihres Betriebssystems ermittelt wird. In den meisten Fällen völlig unproblematisch, aber wenn Ihr Computer auf Deutsch eingestellt ist und Sie beispielsweise mit türkischen Texten arbeiten, dann kann es zu unerwarteten Ergebnissen kommen. Wenn Sie also wissen, in welcher Sprache ein String vorliegt, dann ist es empfehlenswert, auch ein entsprechendes Locale mitzugeben.

```
String geschrei = gerede.toUpperCase(Locale.GERMANY);
```

Listing 8.4 Hier wird in Großbuchstaben geschrien wie in Deutschland.

Locale

Die Klasse Locale dient dazu, Sprachregionen in Java darzustellen. Eine solche Region setzt sich aus ein bis drei Elementen zusammen:

- **Sprache:** Ein zwei- oder dreistelliger Sprachcode nach ISO-639 Teil 1 oder 2 (*http://de.wikipedia.org/wiki/ISO-639*). Häufig gebrauchte Codes sind zum Beispiel »de« für Deutsch, »en« für Englisch und »fr« für Französisch.

- **Land:** ein Ländercode nach ISO-3166 (*http://de.wikipedia.org/wiki/ISO-3166*), entweder ein zweibuchstabiges Länderkürzel wie DE (Deutschland), GB (Großbritannien) oder US (Vereinigte Staaten) oder ein dreistelliger Zahlencode
- **Variante:** ein Freitext, der eine Unterart der Sprache wiedergibt, zum Beispiel Kölsch

Wenn Sie ein `Locale` festlegen, können Sie entscheiden, nur die Sprache anzugeben, beispielsweise de für »deutschsprachig«. Sie können aber zusätzlich auch ein Land angeben, um genauer zu bestimmen, welches Deutsch Sie meinen: de-DE für Deutsch aus Deutschland, de-CH für Deutsch aus der Schweiz … Als Drittes können Sie, um noch genauer zu sein, eine Variante angeben, diese wird aber nur an wenigen Stellen auch ausgewertet, und es gibt keine definierte Liste von Varianten.

Für einige gebräuchliche `Locale` gibt es Konstanten, zum Beispiel `Locale.GERMAN`, aber für die meisten Fälle erzeugen Sie ein neues `Locale`-Objekt mit den entsprechenden Kürzeln:

```
Locale finnisch = new Locale("fi");
```

»startsWith«, »endsWith« und »contains«

Diese drei Methoden prüfen, ob der übergebene Teil-String im String vorkommt. Dabei prüft `startsWith`, ob der String mit diesem Teil-String beginnt, `endsWith`, ob er damit endet, und `contains`, ob er irgendwo im String vorkommt.

```
if (kommentar.toLowerCase().contains("mist")){
    throw new IllegalArgumentException("Fluchen ist hier nicht erwünscht");
}
```

Listing 8.5 Fluchfilter mit »contains«

replace

Anstatt nur zu prüfen, ob ein Teil-String im String vorkommt, können Sie ihn auch gleich durch einen anderen Teil-String ersetzen.

```
kommentar.replace("beschissen", "süß");
```

Listing 8.6 Der Fluchentschärfer peinlich für jeden, der in den Kommentaren flucht

Es gibt häufig Verwirrung mit den beiden Methoden `replace` und `replaceAll`. Auch `replace` ersetzt *alle* Vorkommen des Such-Strings durch den neuen Teil-String. Der Unterschied zwischen den beiden Methoden ist der, dass `replaceAll` den String mit einem regulären Ausdruck durchsucht, um Stellen zu finden, die ersetzt werden sollten. Sie finden deshalb mehr zu dieser Methode in Abschnitt 8.3, »Reguläre Ausdrücke«.

»indexOf« und »lastIndexOf«

Anstatt nur wie `contains` und seine Verwandten zu prüfen, ob ein Teil-String in diesem String vorkommt, geben `indexOf` und `lastIndexOf` auch noch zurück, an welcher Stelle dies das erste Mal der Fall ist. Dabei sucht `indexOf` vom Anfang des Strings und `lastIndexOf` vom Ende. Beide Methoden geben `-1` zurück, falls der gesuchte String nicht gefunden wurde.

Beide Methoden können auch mit einem zweiten Parameter aufgerufen werden, der angibt, an welcher Stelle mit der Suche begonnen werden soll. So können Sie nach und nach alle Vorkommen eines Teil-Strings ausfindig machen.

```
public int zaehleVorkommen(String text, String suchString){
    int index = text.indexOf(suchString);
    int vorkommen = 0;
    while (index != -1){
        vorkommen++;
        index = text.indexOf(suchString, index + 1);
    }
    return vorkommen;
}
```

Listing 8.7 Zählen, wie oft ein Teil-String vorkommt

Im Beispiel wird der String `text` so lange durchsucht, bis keine weiteren Vorkommen von `suchString` mehr gefunden werden. Der Aufruf von `indexOf` mit dem Startindex `index + 1` stellt sicher, dass jedes Vorkommen nur einmal gefunden wird, er beginnt die neue Suche nach dem ersten Zeichen des letzten Treffers.

substring

Die `substring`-Methoden erzeugen einen neuen String aus einem Teil dieses Strings. Die Variante mit einem Parameter liefert einen Teil-String, der an der angegebenen Stelle beginnt und bis zum Ende des Original-Strings reicht:

```
String gluecklich = "unglücklich".substring(2);//"glücklich"
```

Die Variante mit zwei Parametern beginnt den neuen String an der ersten angegebenen Stelle und endet *vor* der zweiten.

```
String glueck = "unglücklich".substring(2, 7); //"glück"
```

Merken Sie sich, und ich hebe das noch einmal hervor, weil ich selbst es bis heute immer wieder nachlesen muss: Das Zeichen am Startindex ist im neuen String enthalten, das Zeichen am Endindex ist es nicht.

trim

Die `trim`-Methode ist eine der einfachsten `String`-Methoden: Sie entfernt alle Leerzeichen an Anfang und Ende des Strings.

join

Die statische `join`-Methode nimmt einen Trenner und ein `String[]` (oder beliebig viele einzelne Strings, dazu später mehr im Abschnitt 10.3 »Variable Parameterlisten«) als Parameter und fügt alle Elemente des Arrays zu einem String zusammen, mit dem Trenner jeweils zwischen zwei Elementen.

```
String[] teile = new String[]{"A", "B", "C"};
String zusammen = String.join(", ", teile);
//zusammen ist "A, B, C"
```

Listing 8.8 Strings zusammenfügen

8.2.3 Übung: Namen zerlegen

Zunächst eine einfache Aufgabe mit Strings: Schreiben Sie eine Methode, die Namen im Format »Nachname, Vorname« entgegennimmt und sie im Format »Vorname Nachname« wieder ausgibt. Vergessen Sie nicht die Testfälle! Denken Sie vor allem darüber nach, wie Sie reagieren, wenn kein Komma oder mehrere Kommata vorkommen. Die Lösung zu dieser Übung finden Sie im Anhang.

8.2.4 Übung: Römische Zahlen I

Für eine etwas schwierigere Übung schreiben Sie eine Methode, die einen String mit einer römischen Zahl entgegennimmt und in einen `int`-Wert umrechnet.

Tabelle 8.2 ruft Ihnen die Symbole römischer Zahlen noch einmal in Erinnerung.

Einer		Fünfer	
Symbol	**Wert**	**Symbol**	**Wert**
I	1	V	5
X	10	L	50
C	100	D	500
M	1.000		

Tabelle 8.2 Die römischen Ziffern

Um die Ziffern zu einer Zahl zusammenzusetzen, werden die entsprechenden Symbole wiederholt und aufsummiert, dabei stehen höhere Werte immer vorne. Zum Beispiel ist die Zahl 123 in römischen Ziffern CXXIII.

Es gibt eine Ausnahme von der strikt aufsteigenden Folge: Ein einzelnes Einersymbol kann vor dem nächsthöheren Einer- oder Fünfersymbol stehen, um eine Subtraktion darzustellen. Die Zahl 4 wird als IV geschrieben: 5 – 1. Diese zusammengesetzten Symbole stehen in der Reihenfolge nach dem zugrunde liegenden Symbol. 293 schreiben Sie demnach als CCXCIII.

Sie haben nun die Wahl zwischen der einfachen und der schwierigen Variante dieser Übung. In der einfachen Variante soll Ihr Programm römische Zahlen in einen `int`-Wert umrechnen.

In der schwierigen Variante soll es außerdem ungültige römische Zahlen erkennen und in diesem Fall einen Fehler werfen. Ungültige römische Zahlen sind solche, die Symbole in der falschen Reihenfolge enthalten (zum Beispiel IIC), zu viele gleiche Symbole enthalten (zum Beispiel CCCCC) oder Symbole enthalten, die nicht zusammen vorkommen dürfen (zum Beispiel CMD). Eine kleine Hilfe für den Anfang: Machen Sie sich zunächst klar, in welcher Reihenfolge die Symbole stehen dürfen. Die Aufgabe ist einfacher zu lösen, wenn Sie Viererwerte als ein Symbol interpretieren und nicht als zwei. Die Lösung zu dieser Übung finden Sie im Anhang.

8.2.5 StringBuilder

Um die Klasse `String` herum gibt es ein kleines Ökosystem weiterer Klassen, die weitere Funktionen zur Verfügung stellen. Eine dieser Klassen ist der `StringBuilder`, dessen Name schon verrät, was seine Aufgabe ist: Er baut Strings.

`StringBuilder` kennt zwei wichtige Methoden, die mit verschiedenen Parametertypen überlagert werden: `append` und `insert`. `append` fügt etwas am Ende des gebauten Strings an, wobei dieses etwas ein anderer String sein kann oder ein beliebiger Primitivtyp oder ein Objekt, an dem dann automatisch `toString` gerufen wird, um es an den String anzuhängen.

```
StringBuilder sb = new StringBuilder();
sb.append("Ein neuer String");
String s = sb.toString();
```

Listing 8.9 Einfaches Beispiel mit »StringBuilder«

`insert` tut fast dasselbe, der neue Text wird aber nicht am Ende angehängt, sondern an einer ebenfalls übergebenen Stelle eingefügt:

```
sb.insert(3, " nicht ganz so");
s = sb.toString(); //s ist jetzt "Ein nicht ganz so neuer String"
```

Listing 8.10 Einfügen mit dem »StringBuilder«

`StringBuilder` folgt einem Muster, das Sie bisher noch nicht kennengelernt haben. Er implementiert ein sogenanntes *Fluent Interface*. Das bedeutet nichts anderes, als dass Sie Aufrufe von `append` und `insert` beliebig verketten können. Das ist praktischer und lesbarer, als elfmal `sb.append(…)` zu schreiben.

```
String vorname, nachname, strasse, hausnummer, plz, stadt;
…
StringBuilder sb = new StringBuilder();
String addresse =
    sb.append(vorname).append(" ")
      .append(nachname).append("\n")
      .append(strasse).append(" ");
      .append(hausnummer).append("\n\n")
      .append(plz).append(" ")
      .append(stadt).toString();
```

Listing 8.11 Das Fluent Interface von »StringBuilder«

Fluent Interface

Ein Fluent Interface, wie `StringBuilder` es bereitstellt, ist keine Hexerei. Es muss einfach nur jede Methode, die am Fluent Interface teilnimmt, `this` zurückgeben. So haben Sie als Ergebnis jedes Methodenaufrufs wieder das Objekt, mit dem Sie gerade arbeiten, und können daran sofort eine weitere Methode aufrufen. Das funktioniert natürlich nur so lange, wie die Methode keinen anderen Rückgabewert haben muss. Im Beispiel muss die Kette nach `toString` enden, denn `toString` gibt einen String zurück.

Fluent Interfaces sind vor allem nützlich für Klassen die wie `StringBuilder` andere Objekte zusammenbauen, denn in diesem Fall müssen Sie häufig viele Methoden aufrufen, um jeden einzelnen gewünschten Wert zu setzen.

Eine Frage, die sich über `StringBuilder` noch stellt, ist das Warum. Sie könnten zumindest die Aufgabe von `append` auch mit reiner String-Konkatenation lösen, und eine `insert`-Methode könnte auch der String zur Verfügung stellen. Warum also den Umweg über `StringBuilder`?

Es geht hier um Speicherverbrauch und Performanz. Bei jeder Konkatenation wird ein neues String-Objekt erzeugt. Das gilt insbesondere auch, wenn Sie mehrere Konkatenationen zu einer Anweisung verketten, etwa so:


```
String adresse = vorname + " " + nachname + "\n" + …;
```

Für jede Konkatenation, also für jeden +-Operator in dieser Anweisung, wird ein neues String-Objekt erzeugt. Das erste enthält den Text `"Vorname "` (beachten Sie das Leerzeichen), das zweite dann `"Vorname Nachname"` usw. Bis auf das Endergebnis werden diese Strings nur benötigt, um die nächste Konkatenation auszuführen, danach werden sie nicht mehr gebraucht. Es entstehen so in kurzer Zeit viele Objekte, die sofort danach wieder freigegeben werden können und dadurch Arbeit für den *Garbage Collector* verursachen.

Der `StringBuilder` vermeidet diesen Müllberg, denn durch `append` wird kein neues Objekt erzeugt, sondern der Inhalt des bestehenden Objekts verändert. Das macht keinen nennenswerten Unterschied, wenn Sie zwei Strings zusammensetzen wollen. Aber bei zehn, 100 oder 1.000 Strings bekommt der Garbage Collector schnell viel Arbeit abgenommen.

»StringBuilder« und »StringBuffer«

Es gibt neben `StringBuilder` noch die Klasse `StringBuffer`, die die gleichen Methoden anbietet und laut Javadoc auch den gleichen Zweck erfüllt. Dies führt häufig zur Verwirrung, denn der Unterschied zwischen beiden Klassen fällt nur auf, wenn Sie das Javadoc sehr aufmerksam lesen.

Ein `StringBuffer` ist dafür geeignet, in mehreren Threads, also Ausführungssträngen des Programms, verwendet zu werden. Er garantiert, dass keine Fehler auftreten, auch wenn zwei Threads gleichzeitig Operationen auf dem `StringBuffer`-Objekt ausführen wollen. Wenn Sie in derselben Situation einen `StringBuilder` verwenden, dann kann eine Vielzahl lustiger Fehler auftreten. Im Gegenzug ist `StringBuilder` aber etwas schneller. Meine Empfehlung ist, immer `StringBuilder` zu verwenden, außer wenn Sie wirklich aus mehreren Threads einen einzigen String zusammenbauen müssen.

8.2.6 Übung: Römische Zahlen II

Erweitern Sie das Programm aus der vorigen Übung. Schreiben Sie eine neue Methode, die einen `int`-Wert mithilfe eines `StringBuffer` in eine römische Zahl umsetzt. Die Lösung zu dieser Übung finden Sie im Anhang.

8.2.7 StringTokenizer

Der `StringTokenizer` ist das Gegenteil des `StringBuilder`. Genau wie der Builder einen String zusammensetzt, zerlegt der Tokenizer ihn. Dazu durchsucht er den String nach Trennzeichen, den sogenannten *Delimitern*, und gibt die Teil-Strings zurück, die

dazwischenliegen. Das tut er allerdings nicht mit einem Mal als Array, sondern nach und nach, einen Teil-String (Token) je Aufruf.

```
StringTokenizer tokenizer = new StringTokenizer("Watson, John, Dr.", ",");
while (tokenizer.hasMoreTokens()){
    System.out.println(tokenizer.nextToken());
}
```

Listing 8.12 Strings zerlegen mit dem »StringTokenizer«

Im Konstruktor übergeben Sie dem Tokenizer den String, auf dem er operieren soll, sowie die Trennzeichen. Als Trennzeichen übergeben Sie zwar nur einen String, aber jedes Zeichen dieses String wird als Trennzeichen interpretiert. Übergeben Sie also `",."`, dann wird an jedem Punkt und an jedem Komma getrennt. Sie können auch keine Trennzeichen übergeben, dann sucht der Tokenizer nach Leerzeichen, Tabulatoren und Zeilenumbrüchen.

Wenn Sie eine Instanz von `StringTokenizer` haben, gibt es zwei interessante Methoden. `hasMoreTokens` prüft, ob ein weiteres Token aus dem String gelesen werden kann, `nextToken` liest dieses Token. Wie im Beispiel gezeigt, sollten Sie immer zuerst prüfen, ob es weitere Token gibt, bevor Sie das nächste Token lesen, denn `nextToken` wirft einen Fehler, wenn keine Token mehr vorhanden sind.

8.3 Reguläre Ausdrücke

Ein großes Thema in der Umgebung von Strings sind *reguläre Ausdrücke* (engl. *regular expressions*). Das sind Muster, die auf Strings angewendet werden können, um beispielsweise zu prüfen, ob ein String einem bestimmten Format entspricht, um einen String in Einzelteile zu zerlegen oder um einen Teil-String darin zu finden, mit mehr Flexibilität, als `indexOf` sie bietet.

8.3.1 Einführung in reguläre Ausdrücke

Die einfachste Anwendung für reguläre Ausdrücke ist die `String`-Methode `matches`, die prüft, ob ein String in seiner Gänze dem Muster entspricht. Ein solches Muster kann dabei einfach nur ein String sein:

```
String eingabe = …;
if (eingabe.matches("abcd")){
    …
}
```

Listing 8.13 Vergleich mit »matches«

Im Beispiel würde `matches true` zurückgeben, wenn `eingabe` den Wert `"abcd"` hat. Das ist noch nichts Besonderes, die `equals`-Methode hätte den gleichen Effekt. Zu einem mächtigen Werkzeug werden reguläre Ausdrücke durch Sonderzeichen, die nicht nur auf genau ein Zeichen im String passen. Dazu gehört zum Beispiel der Punkt, der auf ein beliebiges Zeichen passt. Der reguläre Ausdruck `"abc."` passt (unter anderem) auf die Strings »abcd«, »abc5« und »abc?«, aber nicht auf »abc« oder »abcde«, denn der Punkt passt auf genau ein Zeichen, nicht auf mehrere und auch nicht auf gar keins.

Dazu können Sie *Quantifier* verwenden, Sonderzeichen, die steuern, wie oft ein anderes Zeichen vorkommen darf. Der Stern (*) besagt, dass das vorangehende Zeichen beliebig oft vorkommen darf. Der Ausdruck `"Ka*tze"` passt auf »Katze«, »Kaatze« und »Kaaaaaaatze«, aber auch auf »Ktze«, denn auch 0 Vorkommen sind mit dem Stern erlaubt. Soll das gerade nicht möglich sein, dann können Sie `+` verwenden, womit das vorangehende Zeichen mindestens einmal vorkommen muss, aber auch öfter vorkommen darf. Die Quantifier lassen sich natürlich mit anderen Sonderzeichen wie dem Punkt kombinieren. `eingabe.matches(".+and")` liefert `true` für »Band«, »Rand«, »Sand«, »Brand«, »Strand« etc. Einige weitere Sonderzeichen können Sie Tabelle 8.3 entnehmen.

Zeichen	Bedeutung
`.`	Der Punkt passt auf ein beliebiges Zeichen.
`*`	Das vorangehende Element (ein einzelnes Zeichen, eine geklammerte Zeichenfolge oder eine Zeichenklasse) darf beliebig oft vorkommen, auch keinmal.
`+`	Das vorangehende Element muss mindestens einmal, darf aber auch öfter vorkommen.
`?`	Das vorangehende Element darf nicht oder genau einmal vorkommen, aber keinesfalls öfter.
`{n}`	Das vorangehende Element muss genau n-mal vorkommen.
`{n, m}`	Das vorangehende Element muss mindestens n-mal aber höchstens m-Mal vorkommen.
`^`	Passt auf den Anfang der Eingabe. So passt `^ana` in **ana**nas, aber nicht in b**ana**ne.
`$`	Passt auf das Ende der Eingabe. Zum Beispiel passt `ze$` in Kat**ze**, aber nicht in Kat**ze**n.
`|`	Oder-Verknüpfung, entweder der Ausdruck links oder der Ausdruck rechts muss passen.

Tabelle 8.3 Sonderzeichen in regulären Ausdrücken

Zeichen	Bedeutung
`[abc]`, `[a-z]`, `[a-zA-Z0-9]`	Zeichenklassen passen auf ein Zeichen, das in der Klasse enthalten ist. `[abc]` passt auf a, b oder c, aber nicht auf x. Sie können auch Bereiche angeben, `[a-z]` passt auf alle Kleinbuchstaben. Andere Sonderzeichen verlieren in einer Klasse ihre Wirkung, ein Punkt ist einfach nur ein Punkt, ein Stern nur ein Stern.
`[^abc]`, `[^a-z]`, `[^a-zA-Z0-9]`	So werden Zeichenklassen negiert: `[^abc]` passt auf jedes beliebige Zeichen, außer a, b und c.
`\d`, `\D`	Vordefinierte Zeichenklassen für Ziffern. `\d` passt auf alle Ziffern, entspricht also genau `[0-9]`. `\D` ist genau das Gegenteil und passt auf alles außer Ziffern.
`\s`, `\S`	`\s` ist eine vordefinierte Klasse, die auf Whitespaces passt, also Leerzeichen, Zeilenumbruch, Tabulator usw. `\S` ist wieder das Gegenteil und passt auf alles außer Whitespace.
`\w`, `\W`	`\w` entspricht `[a-zA-Z_0-9]`, `\W` ist die Negation davon.

Tabelle 8.3 Sonderzeichen in regulären Ausdrücken (Forts.)

[!]

Zeichenklassen in Java-Strings

Reguläre Ausdrücke werden in Java, anders als in vielen anderen Sprachen, als Strings angegeben. Das heißt, dass auch alle Regeln für das Escapen von Sonderzeichen gelten wie in jedem anderen String. Problematisch ist das für die vordefinierten Klassen, die mit einem Backslash anfangen, denn der Backslash ist ja gerade das Escape-Zeichen. Deswegen muss er immer gedoppelt werden. Die Klasse `\d` muss also zum Beispiel als `"\\d"` geschrieben werden.

Mit diesen Sonderzeichen können Sie schon viele komplexe Muster ausdrücken, und es gibt noch eine lange Reihe weiterer Zeichen, deren Aufzählung hier zu weit führen würde. Mit regulären Ausdrücken lassen sich viele Dinge einfach umsetzen, die ohne aufwendig wären. Erinnern Sie sich zum Beispiel an die Validierung, ob ein String nur aus Ziffern besteht. Mit einem regulären Ausdruck reduziert sich das auf `string.matches("\d*")`.

8.3.2 String-Methoden mit regulären Ausdrücken

Sie wissen nun, wie Sie einfache reguläre Ausdrücke schreiben. Damit können Sie mehr tun, als nur zu prüfen, ob Strings Ihrem Muster entsprechen. Die `String`-Klasse bietet neben `matches` noch zwei weitere Methoden, die mit regulären Ausdrücken arbeiten.

»replaceAll« und »replaceFirst«

Wie weiter oben bereits angesprochen wurde, ist der Unterschied zwischen `replace` und `replaceAll` nicht, wie viele gefundene Vorkommen ersetzt werden, sondern dass `replaceAll` mit einem regulären Ausdruck festlegt, welche Teil-Strings ersetzt werden sollen. Es werden alle Teil-Strings ersetzt, die auf den regulären Ausdruck passen.

```
String einString = "1234567890";
einString = einString.replaceAll("[13579]", "");
```

Listing 8.14 Zeichen ersetzen mit »replaceAll«

Nach dem gezeigten Code enthält `einString` nur noch die geraden Ziffern: `"24680"`, da alle ungeraden durch einen Leer-String ersetzt wurden. Achten Sie auf die eckigen Klammern im regulären Ausdruck. Sie bilden eine Zeichenklasse mit den fünf enthaltenen Ziffern, dadurch passt jede der Ziffern einzeln auf das Muster.

Aber `replaceAll` kann mehr. Sie können in der Ersetzung den gefundenen Teil-String oder Teile davon wieder einsetzen. Das folgende Codefragment benutzt diese Fähigkeit, um alle Zahlen in einem String in Anführungszeichen zu setzen. Aus `"Die Zahlen 123 und 456"` wird so `"Die Zahlen \"123\" und \"456\""`.

```
einString.replaceAll("\\d+", "\"$0\"");
```

Listing 8.15 Zahlen in Anführungszeichen setzen in einer Zeile

Der reguläre Ausdruck sucht nach allen Vorkommen von einer oder mehr Ziffern. Im Ersetzungs-String bedeutet `$0`, dass an dieser Stelle der komplette gefundene Teil-String eingesetzt werden soll. Es wird so jede Zahl ersetzt durch dieselbe Zahl in Anführungszeichen.

Für noch etwas mehr Feinarbeit können Sie im regulären Ausdruck *Gruppen* bilden, indem Sie Klammern setzen. Der Teil des Strings, der auf diesen Teil des regulären Ausdrucks passt, ist im Ersetzungs-String durch `$1`, `$2`, `$3`, ... verwendbar. `$1` benutzt die erste Gruppe, `$2` die zweite Gruppe usw. Das wird an einem Beispiel etwas klarer. Der reguläre Ausdruck `([0-9+\\-/* ]+)=([0-9+\\-/* ]+)` passt auf einfache Gleichungen wie *17 + 4 = 3 × 7*. In diesem konkreten Fall wäre `$0` = `17 + 4 = 3 * 7`, `$1` = `17 + 4` (die ersten Klammern im regulären Ausdruck) und `$2` = `3 * 7` (die zweiten Klammern im regulären Ausdruck). Jetzt könnten Sie zum Beispiel die beiden Seiten der Gleichung vertauschen:

```
gleichung.replaceAll("([0-9+\\-/*]+)=([0-9+\\-/*]+)", "$2 = $1");
```

Und das Beste daran: Wenn die Gleichung nur Teil eines längeren Strings ist, funktioniert es trotzdem.

split

Die `split`-Methode bietet eine weitere Möglichkeit, einen String zu zerlegen. Im Gegensatz zum `StringTokenizer` können Sie bei `split` aber einen regulären Ausdruck als Trenner angeben. Der String wird an jedem Treffer für den regulären Ausdruck getrennt, und alle so entstandenen Teile werden mit einem Aufruf in einem String-Array zurückgegeben.

```
String katzenString = "Pixie, Garfield, Snowball";
String[] katzen = katzenString.split("\\s*,\\s*");
//katzen = ["Pixie","Garfield","Snowball"]
```

Listing 8.16 Strings trennen mit »split«

Das Trennen am Komma würde auch mit dem `StringTokenizer` funktionieren, aber mit `split` brauchen Sie keine Schleife, und nebenbei werden im Beispiel auch die Leerzeichen beim Trennen entfernt, da sie durch `\\s*` Teil des Trenners sind.

8.3.3 Reguläre Ausdrücke als Objekte

Auch reguläre Ausdrücke sind in Java Objekte. Wenn Sie die String-Methoden verwenden, dann werden diese Objekte zwar vor Ihnen verborgen, aber dennoch sind sie im Hintergrund vorhanden. Und Sie können auch direkt mit den Objekten arbeiten, was in seltenen Fällen notwendig ist, weil Sie mit den `String`-Methoden nicht das erreichen, was Sie möchten.

Zwei Klassen sind im Zusammenhang mit regulären Ausdrücken wichtig: `Pattern` und `Matcher`. Ein `Pattern` ist ein Objekt, das den regulären Ausdruck beinhaltet. Der String, den Sie an die String-Methoden als Parameter übergeben, ist eine textuelle Darstellung des regulären Ausdrucks, aber bevor wirklich etwas damit getan werden kann, wird ein `Pattern` daraus erzeugt. Das können Sie auch selbst tun, bei Pattern geht das allerdings nicht durch einen Konstruktor, sondern durch die statische Methode `Pattern.compile`, der Sie wie gewohnt die String-Repräsentation eines regulären Ausdrucks übergeben.

```
Pattern pattern = Pattern.compile("[0-9]+");
```

Listing 8.17 Ein Pattern-Objekt erzeugen

Das `Pattern`-Objekt ist der anwendbare reguläre Ausdruck. Wenn Sie diesen nun auf einen String anwenden wollen, dann tun Sie das mit der Methode `matcher`, die ein `Matcher`-Objekt erzeugt. Ein `Matcher` ist die konkrete Anwendung des regulären Ausdrucks auf einen bestimmten String. Wenn Sie einen `Matcher` erzeugt haben, dann verwenden Sie entweder dessen Methode `matches`, um zu prüfen, ob der gesamte String zum regulären Ausdruck passt, oder `find`, um Teil-Strings zu finden, die passen.


```
Matcher matcher = pattern.matcher("Die Zahl 12 ist das doppelte von 6.");
while(matcher.find()){
    System.out.println(matcher.group());
}
```

Listing 8.18 Zahlen finden mit »Matcher«

Da mit `find` mehrere Treffer möglich sind, können Sie in einer Schleife alle Vorkommen finden, die zu Ihrem regulären Ausdruck passen. Der nächste Aufruf von `find` beginnt die Suche immer nach dem letzten Treffer, so dass Sie mit jedem Aufruf einen neuen Treffer finden. Die Methode `group` gibt den Text zurück, der zuletzt als Treffer gefunden wurde. Mit dem Pattern von oben gibt dieses Stück Code also nacheinander die Zahlen 12 und 6 aus.

Der große Vorteil von `Pattern` und `Matcher` gegenüber den String-Methoden ist, dass Sie mehr Kontrolle ausüben können. Der wichtigste Anwendungsfall davon ist, dass Sie aus Ihrem Java-Code auf den Inhalt von Gruppen zugreifen können, die Sie in ihrem regulären Ausdruck mit Klammern umgeben haben. Auf diese können Sie zugreifen, indem Sie der `group`-Methode einen `int`-Parameter übergeben: `group(0)` gibt den gesamten Treffer zurück, genau wie `group()` ohne Parameter, `group(1)` den Inhalt der ersten Klammern, `group(2)` den Inhalt der zweiten usw. Damit haben Sie ein sehr vielseitiges Werkzeug an der Hand, um Daten aus einem String zu extrahieren. So kann die Klasse `Preisfinder` zum Beispiel aus einem Text alle Preise extrahieren, die ein dreibuchstabiges Währungskürzel verwenden, also so etwas wie `"500 EUR"`.

```
public class Preisfinder {
    private static final Pattern PREIS_PATTERN =
     Pattern.compile("([0-9.,]+)\\s*([A-Z]{3})");
    private final Matcher matcher;

    public Preisfinder(String suchstring){
        if (suchstring == null){
            throw new IllegalArgumentException("Suchstring darf nicht null sein");
        }
        this.matcher = PREIS_PATTERN.matcher(suchstring);
    }

    public Preis naechsterPreis(){
        if (matcher.find()){
            String wertAlsString = matcher.group(1);
            double wert = Double.parseDouble(wertAlsString);
            String waehrung = matcher.group(2);
            return new Preis(waehrung, wert);
        } else {
```

```
            return null;
        }
    }
}
```

Listing 8.19 Preise finden mit regulären Ausdrücken

In diesem Beispiel nutzen wir aus, dass der `Matcher` sich merkt, von wo er weitersuchen muss. Da der `Matcher` eine Instanzvariable von Preisfinder ist, bleibt er über mehrere Aufrufe von `naechsterPreis` hinweg gleich, und jeder Aufruf gibt das nächste Vorkommen aus.

Was innerhalb der Methode `naechsterPreis` passiert, ist dann keine Hexerei, es wird mit `find` geprüft, ob ein weiterer Preis vorkommt, und dann mit `group(1)` und `group(2)` der Wert der beiden geklammerten Gruppen im regulären Ausdruck ausgelesen. Der Rückgabetyp `Preis` fasst diese beiden Werte lediglich in einem Objekt zusammen.

Das Pattern ist hier als Konstante definiert, weil die Erzeugung des `Pattern`-Objekts aus einem String eine sehr teure Operation ist und nicht mehrmals ausgeführt werden muss, wenn es sich vermeiden lässt.

Benannte Gruppen

Anstatt auf Gruppen durch ihren Index zuzugreifen, können Sie ihnen auch Namen geben. Dadurch wird Ihr Code lesbarer, der reguläre Ausdruck aber länger und unübersichtlicher. Um einer Gruppe einen Namen zu geben, muss in den Klammern an erster Stelle ein Fragezeichen stehen, gefolgt vom Namen in spitzen Klammern, und erst danach der reguläre Ausdruck für die Gruppe. Der reguläre Ausdruck für den Preisfinder sähe dann so aus:

```
(?<wert>[0-9.,]+)\\s*(?<waehrung>[A-Z]{3})
```

Das ist wie gesagt viel weniger übersichtlich, aber im Java-Code wird mit `matcher.group("wert")` sofort viel klarer, was gemeint ist.

8.3.4 Übung: Flugnummern finden

Kennen Sie das? Sie haben einen Flug gebucht, aber wenn Sie in Ihrem Postfach nach der Buchungsbestätigung suchen, dann ist sie nicht aufzufinden. Schreiben Sie eine Methode, die reguläre Ausdrücke benutzt, um ein String-Array – Ihr Postfach – nach der ersten Flugnummer zu durchsuchen. Eine Flugnummer besteht immer aus zwei Großbuchstaben, gefolgt von einem Bindestrich, gefolgt von einer Reihe von Ziffern. Die Lösung zu dieser Übung finden Sie im Anhang.

8.4 Zeit und Datum

Damit ist es für den Augenblick genug von String und seinem Umfeld. Neben Strings und Zahlen sind Zeiten und Daten weitere, häufig benötigte Datentypen.

Die Handhabung von Zeiten und Daten hat sich in Java 8 gegenüber den Vorversionen komplett geändert. Bis Java 7 wurden Zeitwerte durch die Klasse `java.util.Date` abgebildet. Für komplexere Operationen mit Datums- und Zeitwerten gab es die Klasse `java.util.Calendar`, wegen der undurchsichtigen und übermäßig abstrakten API eine der unbeliebtesten Klasse in Javas Standardbibliothek.

Mit Java 8 wird alles anders, und alles, was mit Zeit zu tun hat, wird von den Klassen aus dem Package `java.time` gehandhabt.

8.4.1 Zeiten im Computer und »java.util.Date«

Zeit und Datum werden nicht nur in Java, sondern von Computern ganz allgemein meistens als eine einfache Zahl dargestellt: die Zahl von Millisekunden, die seit dem 1.1.1970, 00:00:00 vergangen sind. Dazu wird heute normalerweise ein 64-Bit-Wert verwendet, in Java ein `long`, der für die nächsten Millionen Jahre noch ohne Überlauf funktionieren wird. Sie können die aktuelle Zeit in dieser Form durch die Methode `System.currentTimeMillis` auslesen.

Ein `Date` ist nichts anderes als eben ein solcher `long`-Wert, verpackt in einem Objekt und mit einigen Methoden angereichert, zum Beispiel `before` und `after`, die prüfen, ob ein anderes Datum früher oder später liegt. Es gibt zwar viele Methoden, die aus dieser einen Zahl eine lesbarere Aussage machen (`getMinutes`, `getHours`, ...), sie sind aber *deprecated* und sollten deshalb nicht mehr verwendet werden.

Die allgemeine Empfehlung lautet sowieso, dass Sie die neuen Klassen aus dem `java.time`-Package verwenden sollten. Das ist in der Theorie eine gute Idee, in der Praxis benötigen Sie aber nach wie vor häufig ein `java.util.Date`. Deshalb ist es gut, dass `Date` seit Java 8 die statische Methode `from` hat, die aus einem `java.time.Instant` – gleich mehr dazu – ein gutes, altes `Date`-Objekt erzeugt. So können Sie die neuen Klassen nutzen und dennoch mit alten Methoden kommunizieren, die noch `Date` erwarten.

8.4.2 Neue Zeiten – das Package »java.time«

Die neuen Klassen aus `java.time` sind etwas komplexer, als `Date` es war. Und es gibt viel mehr von ihnen. Die vielen neuen Klassen lösen aber mehrere Probleme, die mit `Date` immer wieder zu Schwierigkeiten geführt haben. Bei einem `Date`-Objekt war nie klar, in welche Zeitzone es gehörte. Für viele Anwendungen ist das kein Problem, aber wenn Sie einem Benutzer die Abflugzeit seines Fluges anzeigen wollen, dann ist es

schon wichtig, zu unterscheiden, ob dies die Zeit in seiner Zeitzone oder in der Zeitzone des Flughafens oder vielleicht in UTC (koordinierte Weltzeit) ist. Die neuen Klassen handhaben das unkompliziert.

Es gab außerdem in Java keine standardisierte Möglichkeit, Angaben zu machen, bei denen bestimmte Teile nicht gefüllt waren. Zum Beispiel konnten Sie nicht Jahr und Monat angeben, ohne auch den Tag zu setzen, oder Monat und Tag ohne das Jahr – in einem `Date` steckte immer alles drin. Sie konnten niemals Ihren Geburtstag angeben, ohne das Jahr zu verraten. Das ist seit Java 8 möglich. Der Preis dafür ist, dass es jetzt einen kleinen Zoo voller Klassen gibt. Tabelle 8.4 verschafft Ihnen einen Überblick.

Klasse	Beschreibung
`Instant`	Die Klasse `Instant` entspricht am ehesten der alten Klasse `Date`, ein `Instant`-Objekt steht für einen Moment auf dem Zeitstrahl, mit Genauigkeit im Millisekunden- oder sogar Nanosekundenbereich, abhängig von Hardware und Betriebssystem.
`LocalDate`	Speichert ein Datum (Tag, Monat und Jahr).
`LocalTime`	Speichert eine Zeit (Stunden, Minuten, Sekunden, Milli- und Nanosekunden). `LocalTime` berücksichtigt keine Zeitzonen, Werte sollten als Ortszeit interpretiert werden.
`LocalDateTime`	Kombiniert `LocalDate` und `LocalTime` für eine Datums- und Zeitangabe.
`ZonedDateTime`	Eine kombinierte Datums- und Zeitangabe mit Zeitzone. Die Zeitzone wird in Form einer `ZoneId` angegeben.
`Month`	Der enumerierte Datentyp `Month` stellt einen Monat dar.
`Year`	eine Jahresangabe
`YearMonth`	Angabe von Jahr und Monat, ohne Tag oder Uhrzeit, zum Beispiel das Gültigkeitsdatum Ihrer Kreditkarte
`MonthDay`	Monat und Tag, ohne Jahresangabe, zum Beispiel Ihr Geburtstag, wenn Sie Ihr Alter nicht verraten möchten
`DayOfWeek`	ein weiterer enumerierter Typ für die Wochentage
`Duration`	Duration gibt als einzige der hier vorgestellten Klassen keinen Zeitpunkt an, sondern eine Dauer, also eine Angabe wie »2 Tage und 12 Stunden«.

Tabelle 8.4 Die wichtigsten Klassen aus »java.time«

Das sind zwar viele Klassen, aber sie haben alle eine konsistente API, die es erleichtert, den Überblick zu behalten. Dazu kommen über alle Klassen hinweg Methoden mit gleichen Präfixen zum Einsatz:

- `of`-: Methoden mit Namen oder Präfix `of` dienen dazu, neue Instanzen der jeweiligen Klasse zu erzeugen. Die `java.time`-Klassen bieten keine öffentlichen Konstruktoren, sondern nur die statischen `of`-Methoden zu diesem Zweck.

  ```
  LocalDate starWarsRelease = LocaleDate.of(1977, Month.MAY, 25);
  MonthDay weihnachten = MonthDay.of(Month.DECEMBER, 24);
  ```

- `now`: Die `now`-Methoden liefern immer ein Objekt der jeweiligen Klasse, das die aktuelle Zeit enthält.

  ```
  Instant genauJetzt = Instant.now();
  LocalDate heute = LocalDate.now();
  ```

- `get`-: Getter haben dieselbe Funktion, die Sie schon ausführlich kennengelernt haben.
- `with`-: Erzeugt ein neues Objekt, bei dem der Wert eines Feldes geändert wurde. Alle Zeit-Objekte sind unveränderlich, deshalb gibt es keine Setter. Diese Rolle wird von den `with`-Methoden übernommen, die ein neues Objekt mit diesem Wert zurückgeben.

  ```
  LocalDate heute = LocalDate.now();
  LocalDate monatsanfang = heute.withDayOfMonth(1);
  ```

- `plus`-, `minus`-: Diese Methoden addieren zum oder subtrahieren vom Wert dieses Objekts. Die `with`-Methoden setzen einen neuen Wert, `plus`- und `minus`- berechnen ihn.

  ```
  LocalDate heute = LocalDateNow();
  LocalDate naechsteWoche = heute.plusWeeks(1);
  ```

- `at`-: Kombiniert dieses Objekt mit einem anderen Zeit-Objekt. Aus einem `MonthDay` wird so mit `atYear` ein `LocalDate`: Zu Monat und Tag kommt ein Jahr hinzu, es entsteht eine vollständige Datumsangabe.

  ```
  MonthDay meinGeburtstag = MonthDay.of(Month.DECEMBER, 13);
  LocalDate diesesJahr = meinGeburtstag.atYear(2014);
  DayOfWeek wochentag = diesesJahr.getDayOfWeek();
  System.out.println("Dieses Jahr ist es ein " + wochentag);
  ```

Über diese Methoden hinaus haben alle Klassen noch diverse `boolean`-Methoden mit dem Präfix `is`, die verschiedene Prüfungen implementieren. `isBefore` und `isAfter` prüfen, ob eine andere Zeitangabe einen früheren oder späteren Zeitpunkt angibt. Zeitangaben, die ein Jahr enthalten, können mit `isLeapYear` prüfen, ob es sich um ein Schaltjahr handelt.

Details, welche Methoden aus den jeweiligen Präfixgruppen eine Klasse implementieren, entnehmen Sie am besten dem Javadoc.

Alles sehr eurozentrisch?

Alle oben vorgestellten Zeit- und Datumsklassen verwenden den gregorianischen Kalender, also den hier völlig normalen Kalender, den Sie jeden Tag benutzen. Dieser Kalender ist aber keineswegs weltweiter Standard, es werden auch heute noch verschiedene andere Kalendersysteme verwendet, die sich mehr oder weniger von »unserem« Kalender unterscheiden. Was ist mit denen?

In Java wurde schon immer versucht, nicht nur auf die Lokalitäten der westlichen Welt einzugehen, sondern auch andere Regionen zu unterstützen. So entstand zum Beispiel die alte `Calendar`-Klasse, deren Hauptproblem es war, dass sie mit einem Interface alle möglichen Kalendersysteme abstrahieren wollte. Und dann war alles so abstrakt, dass es nicht mehr bedienbar war und alle nur noch die Spezialisierung `GregorianCalendar` verwendet haben.

Mit dem neuen Ansatz wurde deswegen nicht versucht, alles gleichwertig abzubilden. Der gregorianische Kalender (richtiger der Kalender nach ISO-8601) hat sich für die internationale Kommunikation als Standard durchgesetzt, deshalb ist es verständlich, dass er auch in Java eine herausgestellte Position einnimmt.

Das heißt aber nicht, dass andere Kalender nicht unterstützt werden. Im Package `java.time.chrono` gibt es Datumsklassen, die verschiedene andere Kalender unterstützen (Hijrah, Minguo, ...). Dort liegt auch die abstrakte Klasse `Chronology`, die Sie erweitern können, um weitere Kalendersysteme zu realisieren. Zwischen allen verschiedenen Arten von Datumswerten können Sie konvertieren, indem Sie die Methode `from(TemporalAccessor)` des Zieldatentyps rufen. Alle Zeit- und Datumstypen implementieren das Interface `TemporalAccessor` und sind deshalb gültige Parameter.

Zeiten mit Zeitzonen

Zeitzonen fügen eine weitere Ebene an Komplexität zu allem hinzu, was mit Zeiten zu tun hat. Das ist auch in Java 8 nicht zu vermeiden, Zeitzonen sind nun einmal komplex. Aber vieles wird bereits dadurch erleichtert, dass es jetzt verschiedene Klassen für Zeiten mit und ohne Zeitzone gibt.

Werte mit Zeitzone verwenden die Klasse `ZonedDateTime`, die zusätzlich zu Datum und Zeit noch eine `ZoneId` enthält, einen Bezeichner für eine Zeitzone. Es gibt zwei Arten von `ZoneId`, die Sie alle durch die Methode `ZoneId.of` erzeugen können:

- Offsets geben den Abstand der Zeitzone zu UTC an. Dieser Abstand ist fest, Besonderheiten wie Sommer- und Winterzeit können nicht berücksichtigt werden. In der Zeitzone `ZoneId.of("+2")` ist es immer zwei Stunden später als UTC.

- Geografische Regionen geben die Zeit an, wie sie in dieser Region gilt. Das beinhaltet den Offset, aber auch die Umstellung von Sommer- auf Winterzeit und andere eventuelle Besonderheiten. Die Bezeichnung einer solchen Zone besteht immer aus der Region und einer Stadt in dieser Region. Die deutsche lokale Zeitzone erhalten Sie mit `ZoneId.of("Europe/Berlin")`. Oder Sie können `ZoneId.systemDefault()` rufen und damit die Zeitzone bekommen, die im Betriebssystem konfiguriert ist. Es gibt eine sehr lange Liste von unterstützten Zeitzonen, die sich mit Java-Updates ändern kann. Alle möglichen Zeitzonen erhalten Sie mit `ZoneId.getAvailableZoneIds()`. Die Methode gibt ein `Set` zurück; wie Sie damit umgehen, erfahren Sie in Kapitel 10, »Arrays und Collections«.

Mit **`ZonedDateTime`** können Sie einfach eine Zeit mit Angabe der Zeitzone erzeugen und mit dieser dann so arbeiten wie mit `LocalDateTime`. Insbesondere funktionieren auch die Methoden `isBefore` und `isAfter` mit `ZonedDateTime` und berücksichtigen die Zeitzonen für den Vergleich. So ist 21:00 in der Zeitzone »Europe/Berlin« früher als 19:00 in der Zeitzone »America/Chicago«, genau wie es sein sollte.

`ZonedDateTime` macht auch die Umwandlung von einer Zeitzone in eine andere einfach. Dazu gibt es zwei Methoden:

- `withZoneSameInstant` gibt dieselbe absolute Zeit in einer anderen Zeitzone an. Aus der Zeit »15.6.2014 12:00 Europe/Berlin« wird durch `withZoneSameInstant("America/Sao_Paulo")` »15.6.2014 07:00 America/Sao_Paulo«. Die beiden Angaben sind gleichzeitig. Das Datum ist bei der Umwandlung relevant, da Sommer- und Winterzeit berücksichtigt werden. Im Dezember beträgt der Unterschied zwischen den Ortszeiten nur 3 Stunden statt 5.
- `withZoneSameLocal` erzeugt ein Objekt, das zwar eine andere Zeitzone enthält, aber die gleiche Ortszeit. Am selben Beispiel: aus »15.6.2014 12:00 Europe/Berlin« wird durch `withZoneSameLocal("America/Sao_Paulo")` »15.6.2014 12:00 America/Sao_Paulo«. Beide Angaben enthalten dieselbe Ortszeit, sind aber **nicht** zeitgleich.

8.4.3 Übung: Der Fernsehkalender

Ich bin ein großer Fan von Fernsehserien aus aller Welt. Den Überblick zu behalten, wann genau eine neue Folge erscheint, ist nicht ganz einfach, denn alles wird in verschiedenen Zeitzonen ausgestrahlt. Da soll nun ein Programm helfen.

Schreiben Sie zunächst zwei Datenklassen. Die Klasse `Fernsehserie` enthält den Namen einer Serie, wie viele Folgen es in dieser Staffel gibt, und zu welcher Zeit die erste Folge in ihrer Ursprungsregion ausgestrahlt wird. Die Klasse `Folge` enthält wiederum den Namen der Serie, die Nummer dieser Folge und Ausstrahlungsdatum und -zeit dieser Folge in Ortszeit.

Schreiben Sie dann die Klasse `Fernsehkalender` mit der statischen Methode `erzeugeKalender`, die ein Array von Fernsehserien als Parameter erhält. Aus diesem Array soll sie ein Array von Folgen berechnen, die alle Folgen all dieser Fernsehserien beinhalten, mit der Ausstrahlungszeit in **Ihrer** Zeitzone. Für die Berechnung wann eine Folge ausgestrahlt wird, nehmen Sie an, dass die Ausstrahlung völlig regelmäßig ist: jede Woche zur gleichen Zeit, und ohne Unterbrechungen. Die Ausgabe soll nach Ausstrahlungsdatum der Folgen sortiert sein. Die Lösung zu dieser Übung finden Sie im Anhang.

8.5 Internationalisierung und Lokalisierung

Wie Sie schon am Umgang mit Zeitzonen und der Option, mit anderen Kalendern zu arbeiten, sehen können, haben sich die Entwickler von Java einige Gedanken darüber gemacht, wie Programme möglichst einfach an verschiedene regionale Anforderungen anzupassen sind. Der Umgang mit Zeit und Datum ist aber nur ein Teil davon, es gibt in der Standardbibliothek noch einige weitere Klassen, die Ihnen erleichtern sollen, Ihre Programme weltweit einsetzbar zu machen. Vielleicht keine Anforderung für kleine Beispielprogramme, aber bei Programmen, die in der realen Welt eingesetzt werden, ist es heute ein wichtiger Teil.

I18N und L10N

Um eine Anwendung in eine andere Sprache zu übersetzen, sind zwei Schritte notwendig: *Internationalisierung* und *Lokalisierung*. Internationalisierung bereitet eine Anwendung darauf vor, in verschiedene Sprachen übersetzt zu werden. Dazu gehört zum Beispiel, dass im Code keine Texte mehr stehen, die ausgegeben werden. Diese Texte stehen in einer Übersetzungsdatei, im Code steht nur noch ein Schlüssel, mit dem in dieser Datei der richtige Eintrag gefunden wird.

Bei der Lokalisierung werden die Ressourcen angelegt, die eine so vorbereitete Anwendung benötigt, um in einer bestimmten Sprache ausgeführt zu werden. Zum Beispiel wird eine Übersetzungsdatei für diese Sprache angelegt. Sie werden sehen, dass Java Ihnen bei der Lokalisierung schon einige Arbeit abnimmt. Sie müssen sich zum Beispiel nicht selbst darum kümmern, dass Zeit und Datum im richtigen Format dargestellt werden.

Häufig liest man statt Internationalisierung und Lokalisierung auch die Abkürzungen I18N und L10N. Sie stehen für I + 18 Buchstaben + N und L + 10 Buchstaben + N, abgekürzt von den englischen Begriffen **I**nternationalisatio**N** und **L**okalisatio**N**.

8.5.1 Internationale Nachrichten mit »java.util.ResourceBundle«

Der größte Teil der Lokalisierung ist ohne Zweifel die Übersetzung angezeigter Texte. Dazu müssen (und sollten) Sie nicht im Code für jeden Text entscheiden, in welcher Sprache er angezeigt werden soll. Es ist viel sauberer und vor allem einfacher, all diese Texte in eine Datei auszulagern und sie von dort zu laden und anzuzeigen. So bleibt Ihr Code klar, und Sie können Übersetzungen in andere Sprachen hinzufügen, ohne dass Sie Änderungen am Programm machen müssen.

Das Dateiformat, das dafür verwendet wird, ist das Properties-Format. Es handelt sich dabei um ein Textformat, bei dem in jeder Zeile ein Schlüssel-Wert-Paar steht. Wenn Sie Properties-Dateien für Übersetzungen benutzen, dann ist der Wert der anzuzeigende Text, den Schlüssel verwenden Sie in Ihrem Code, um auf den Text zuzugreifen. Damit das funktioniert, muss jeder Schlüssel natürlich eindeutig sein.

```
zahleingeben=Bitte geben Sie eine Zahl ein
zugross=Diese Zahl ist zu groß
#Diese Zeile ist ein Kommentar
umbruch=Ein Backslash funktioniert auch hier \
als Escape, zum Beispiel wenn Sie Zeilenumbrüche in \
Ihrem Text benötigen
```

Listing 8.20 Eine Properties-Datei mit Übersetzungen

Aus einer solchen Properties-Datei können Sie ein `ResourceBundle` laden, ein Objekt, das die Übersetzungen aus der Datei enthält. Sie erzeugen ein `ResourceBundle` mit der statischen Methode `ResourceBundle.getBundle`. Als Parameter übergeben Sie den Namen des Bundles und, falls nicht das Default-`Locale` des Systems verwendet werden soll, ein `Locale`. Aus den Parametern ergibt sich der Dateiname der Properties-Datei, die geladen wird.

Damit eine Datei überhaupt mit dem Standardmechanismus geladen werden kann, muss sie im Klassenpfad liegen, also direkt neben den CLASS-Dateien Ihrer Anwendung. Aber welche Datei wird geladen? Dabei werden die verschiedenen Ausprägungen von `Locale`-Objekten wichtig. Es wird zunächst versucht, eine Datei zu laden, deren Namen sich aus dem übergebenen Namen und `Locale` zusammensetzt – oder dem Default-`Locale`, falls Sie keins übergeben haben. Wollen Sie also das Bundle mit dem Namen Texte im `Locale` de-DE-koelsch laden, dann werden nacheinander diese Dateien gesucht:

- *Texte_de_DE_koelsch.properties*
- *Texte_de_DE.properties*
- *Texte_de.properties*
- *Texte.properties*

Die Suche nach einer passenden Datei endet nicht, wenn eine gefunden wird. Es werden alle passenden Dateien geladen und ihre Inhalte zusammengeführt. Speziellere Dateien können Texte aus allgemeineren Dateien überschreiben. So können Sie deutsche Texte in *Texte_de.properties* speichern, und wenn einige Texte in Deutschland und Österreich unterschiedlich sind, können Sie diese in Dateien namens *Texte_de_DE.properties* bzw. *Texte_de_AT.properties* eintragen, ohne dass Sie auch alle anderen Texte dorthin kopieren müssten.

Wenn Sie ein `ResourceBundle`-Objekt haben, dann holen Sie Texte einfach daraus hervor, indem Sie den Schlüssel des gewünschten Textes an die `getString`-Methode übergeben:

```
public class ResourceBundles {
    private static final ResourceBundle TEXTE =
     ResourceBundle.getBundle("texte");
    public static void main(String[] args) {
        …
        System.out.println(TEXTE.getString("beispiel"));
    }
}
```

Listing 8.21 »ResourceBundle« laden und verwenden

In den meisten Fällen sollten Sie ein `ResourceBundle` auch wie gezeigt verwenden. Sie können es als Konstante deklarieren, weil sich die Sprache nicht ändert, während das Programm läuft, und Sie müssen kein `Locale` übergeben, denn dann wird die Systemsprache benutzt, die meistens die richtige für den Benutzer ist.

8.5.2 Nachrichten formatieren mit »java.util.MessageFormat«

In vielen Fällen wollen Sie aber nicht nur einen gleichbleibenden String anzeigen, sondern Daten aus Ihrem Programm in den String einbauen: "`Am` **`16.05.2014`** `geht die Sonne in` **`Bonn`** `um` **`5:38`** `Uhr auf.`" Sie könnten zwar die Satzteile zwischen den fett gedruckten Daten einzeln im `ResourceBundle` definieren und im Code mit den Daten zusammensetzen, aber zu empfehlen ist es nicht. Zum einen ist der resultierende Code nicht besonders schön, aber schwerer wiegt, dass Sie vielleicht die Reihenfolge ändern möchten (oder in einer Sprache müssen), in der die Daten in den Ausgabetext eingefügt werden. Vielleicht möchten Sie den Text in "`In` **`Bonn`**`, am` **`16.05.2014`**`, geht die Sonne um` **`5:38`** `Uhr auf`" ändern. Mit der String-Konkatenation müssten Sie dazu den Code ändern. Das geht besser, und zwar mit `MessageFormat`.

`MessageFormat` erzeugt aus einem String mit Platzhaltern der Form `{0}`, `{1}` usw. einen Ausgabe-String, in dem die Platzhalter durch Parameter ersetzt werden.


```
String ausgabe =
 MessageFormat.format("Am {0} geht die Sonne in {1} um {2} Uhr auf",
 datum, ort, zeit);
```

Listing 8.22 Ausgabe formatieren mit »MessageFormat«

Der Platzhalter `{0}` wird durch den ersten Parameter nach dem Format-String ersetzt, also die Variable `datum`, der Platzhalter `{1}` durch den nächsten Parameter `ort` und `{3}` durch `zeit`. Sie können der `format`-Methode beliebig viele Parameter übergeben – das funktioniert mit einer variablen Parameterliste (siehe Abschnitt 10.3) –, aber es sollten mindestens so viele sein, wie Sie auch Platzhalter verwenden. Die Platzhalter müssen nicht in der richtigen Reihenfolge vorkommen, es kommt nur auf die angegebene Zahl an. Und selbstverständlich kann der Format-String auch aus einem `ResourceBundle` gelesen werden, so dass er inklusive Platzhalter übersetzt werden kann.

Wie gezeigt ruft `MessageFormat` an den übergebenen Parametern `toString`, um sie in die Ausgabe einzufügen. Das ist meistens auch das gewünschte Verhalten, aber nicht für Zahlen, und Datums-/Zeitwerte, denn für diese gibt es verschiedene Arten, wie sie formatiert werden sollen, die auch wieder vom `Locale` abhängen.

Zahlen formatieren in »MessageFormat«

Wenn ein Zahlenwert in Ihrem `MessageFormat` ausgegeben werden soll, dann markieren Sie den entsprechenden Platzhalter als `{0,number}`. Das reicht schon aus, um die Zahl sauber formatiert anzuzeigen, mit nicht zu vielen Nachkommastellen, Gruppierung und den zum `Locale` passenden Gruppierungs- und Dezimalzeichen. Wenn Sie nur eine Zahl nach diesem Muster formatieren möchten, können Sie mit `NumberFormat.getInstance()` auch ein Objekt bekommen, das genau das leistet.

Sie können aber über die Standardformatierung hinaus noch weiter beeinflussen, wie die Ausgabe aussehen soll (siehe Tabelle 8.5).

Muster	Bedeutung
`{0,number,integer}`	Gibt die Zahl ohne Nachkommastellen aus. Entspricht `NumberFormat.getIntegerInstance()`.
`{0,number,currency}`	Verwendet ein spezielles Format für Preisangaben. Entspricht `NumberFormat.getCurrenyInstance()`.
`{0,number,percent}`	Verwendet ein spezielles Format für Prozentangaben. Entspricht `NumberFormat.getPercentInstance()`.

Tabelle 8.5 Die verschiedenen Varianten von Zahlenformaten

Muster	Bedeutung
{0,number,<Pattern>}	Mit dieser Variante können Sie ein eigenes Muster angeben, wie die Zahl formatiert werden soll. Wie genau dieses Muster aussieht, können Sie dem Javadoc der Klasse DecimalNumberFormat entnehmen.

Tabelle 8.5 Die verschiedenen Varianten von Zahlenformaten (Forts.)

Datumswerte formatieren

Leider hat MessageFormat nicht mit den Neuerungen von Java 8 mitgehalten. Sie können zwar auch Datumswerte und Uhrzeiten im Format-String angeben, so dass sie formatiert werden, das funktioniert aber nur mit dem alten java.util.Date und nicht mit den neuen Klassen aus java.time.

Mit einem Date-Objekt können Sie verfahren genau wie mit einer Zahl. Sie geben als Platzhalter entweder {0,date} oder {0,time} an, je nachdem, ob Sie den Datums- oder Zeitanteil ausgeben wollen. Es gibt keine Möglichkeit, beides zusammen auszugeben, in dem Fall müssen Sie im Format-String "{0,date} {0,time}" angeben. Auch diese Werte werden je nach Locale anders formatiert, und auch hier können Sie noch feiner steuern, wie die Ausgabe aussehen soll (siehe Tabelle 8.6).

Muster	Beispiel Datum	Beispiel Zeit
{0,date,short}, {0, time, short}	17.05.14	16:57
{0,date,medium}, {0, time, medium}	17.05.2014	16:57:08
{0,date,long}, {0, time, long}	17. Mai 2014	16:57:08 MESZ
{0,date,full}, {0, time, full}	Samstag, 17. Mai 2014	16:57 Uhr MESZ
{0,date,<Pattern>}, {0, time, <Pattern>}	In dieser Variante können Sie ein eigenes Muster angeben, wie die Ausgabe formatiert werden soll. Details zu diesem Muster finden Sie im Javadoc der Klasse SimpleDateFormat.	

Tabelle 8.6 Datums- und Zeitformate, Beispiele aus dem deutschen »Locale«

Mit den neuen Klassen aus java.time lässt sich nicht ganz so komfortabel umgehen, Sie können nicht direkt im Format-String angeben, wie sie formatiert werden sollen. Sie können aber mit der Klasse DateTimeFormatter jedes der neuen Objekte in einen String formatieren und diesen als Parameter an MessageFormat übergeben.

Die einfachste Möglichkeit dazu ist auch hier wieder, anzugeben, ob Sie ein kurzes oder ausführliches Format wünschen. Das tun Sie mit den statischen Methoden DateTimeFormatter.ofLocalizedDate, ofLocalizedTime und ofLocalizedDateTime. Alle diese Methoden erwarten einen Wert der Enumeration FormatStyle als Parameter: SHORT, MEDIUM, LONG oder FULL, analog zu den Formatangaben für java.util.Date in der Tabelle.

Um mit dem DateTimeFormatter zu einem String zu kommen, rufen Sie die format-Methode eines Datumsobjekts auf und übergeben den Formatter.

```
LocalDateTime termin = …;
DateTimeFormatter format =
 DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);
String text =
 MessageFormat.format("Termin um {0} Uhr!", termin.format(format));
```

Listing 8.23 »java.time«-Objekte in einem »MessageFormat« ausgeben

Sie haben so im Format-String leider weniger Kontrolle über das Ausgabeformat, als Sie es mit java.util.Date hätten. Sie müssen das Datumsformat entweder im Code festlegen oder in Ihrem ResourceBundle einen weiteren Eintrag machen, in dem das Datumsformat definiert wird.

8.5.3 Zeiten und Daten lesen

Daten und Zeiten auszugeben ist aber nur eine Hälfte des Problems und typischerweise die leichter zu lösende. Häufig müssen Sie solche Werte auch aus einem String, meist einer Benutzereingabe, herstellen. Dieser Vorgang heißt *Parsing*.

Die Formatierungsklassen SimpleDateFormat und DateTimeFormatter bieten beide parse-Methoden, die versuchen, ein Objekt aus einem String zu erzeugen. Sie können auch dazu die vordefinierten Formate (short, full, ...) verwenden, aber es ist häufig auch sinnvoll, ein eigenes Format festzulegen, so dass Sie genau vorgeben können, wie Ihr Benutzer seine Eingabe machen soll.

Für beide Klassen geben Sie das Muster, nach dem geparst werden soll, als String an, der festlegt, welcher Teil des Datums an welcher Stelle des geparsten Strings steht. Diese Muster sehen für SimpleDateFormat und DateTimeFormatter ähnlich aus, letztere Klasse unterstützt aber mehr verschiedene Symbole. Im Detail soll nur der neuere DateTimeFormatter betrachtet werden.

Ein Format-String, um Datum oder Uhrzeit einzulesen, besteht aus Folgen von Klein- und Großbuchstaben, die angeben, welches Feld (Tag, Monat, Stunde, ...) an dieser Stelle steht, und anderen Zeichen, die die Felder voneinander trennen. Das übliche deutsche Datumsformat wird so ausgedrückt: dd.MM.yyyy. Das kleine d steht für

den Tag im Monat, das große M für den Monat im Jahr (Groß- und Kleinschreibung sind hier wichtig, das kleine m steht für Minuten) und das kleine y für die Jahreszahl. Aus dem Format-String erzeugen Sie mit `DateTimeFormatter.ofPattern` eine Instanz des Formatters, die Sie an die `parse`-Methode einer Datumsklasse übergeben.

```
System.out.println("Bitte geben Sie Ihr Geburtsdatum im Format TT.MM.JJJJ ein");
String eingabe = in.readLine();
LocalDate geburtstag =
 LocalDate.parse(eingabe, DateTimeFormatter.ofPattern("dd.MM.yyyy",
 Locale.GERMANY));
```

Listing 8.24 Geburtstage einlesen

In diesem Beispiel ist es nicht notwendig, ein `Locale` für `ofPattern` anzugeben. Wenn Sie aber ein Muster mit ausgeschriebenen Monats- oder Tagesnamen verwenden (siehe Tabelle 8.7), dann werden diese in der Sprache des `Locale` erwartet.

Symbol	Erklärung
yy, yyyy	Die Jahreszahl. `yy` gibt die letzten zwei Stellen der Jahreszahl aus, `yyyy` die vollständige Jahreszahl. Beim Parsen verhält es sich genauso, aber eine zweistellige Zahl wird immer als 20xx interpretiert.
MM, MMM, MMMM	Der Monat. `MM` steht für die Monatszahl, `MMM` für die Abkürzung des Monatsnamens (Jan, Feb, ...) und `MMMM` für den ausgeschriebenen Monatsnamen. `MMM` und `MMMM` sind beide vom `Locale` abhängig und machen Ausgaben in der korrekten Landessprache.
dd	der Tag im Monat
EE, EEEE	Der Wochentag. `EE` gibt die Abkürzung (Mon, Die, ...) aus, `EEEE` den vollständigen Tag. Auch diese Ausgabe ist abhängig vom `Locale`.
HH	Die Stunde als 24-Stunden-Angabe. Für die im englischsprachigen Raum übliche 12-Stunden-Angabe verwenden Sie `KK` und aa für die Angabe `AM`/`PM`.
mm	Minuten
ss	Sekunden

Tabelle 8.7 Unterstützte Symbole von »DateTimeFormatter« (Auszug)

8.5.4 Zahlen lesen

Sie kennen bereits Methoden, um Zahlen aus Strings zu lesen: die `parse`-Methoden der jeweiligen Wrapper-Klassen, zum Beispiel `Integer.parseInt` oder `Double.parseDouble`. Für viele Fälle sind diese Methoden ausreichend, sie haben aber zwei kleine Nachteile:

- Sie müssen vorher wissen, welchen Ausgabetyp Sie benötigen. Dieser Nachteil wiegt meist weniger schwer, denn wenn Sie das Ergebnis einer Variablen zuweisen möchten, diktiert diese sowieso, welchen Zahlentyp Sie benötigen.
- Diese Methoden nehmen keine Rücksicht auf Internationalisierung. Das ist ein größeres Problem, vor allem bei `float`- und `double`-Werten, denn `Double.parseDouble` kann mit dem Komma als Dezimaltrenner nichts anfangen. Die nach deutschem Muster formatierte Zahl 123,45 kann nicht gelesen werden, nur die englische Variante 123.45 funktioniert. Ähnliche Probleme können aber auch bei Ganzzahlen auftreten, wenn diese fein säuberlich in Gruppen von drei Ziffern zerteilt sind: 123.456.789 kann von `Integer.parseInt` nicht gelesen werden.

Meist wollen Sie Ihren Benutzern den Komfort bieten, Zahlen in ihrem gewohnten Format eingeben zu können und nicht nur für Ihr Programm einen Punkt statt eines Kommas verwenden zu müssen. Diesen Komfort können Sie mit `NumberFormat` bieten. Instanzen dieser Klasse erhalten Sie durch ihre diversen statischen Factory-Methoden:

- `NumberFormat.getInstance` liefert ein allgemeingültiges Format.
- `NumberFormat.getIntegerInstance` liefert eine Instanz speziell für Ganzzahlen.
- `NumberFormat.getCurrencyInstance` liefert eine Instanz speziell für Währungsangaben.
- `NumberFormat.getPercentInstance` liefert eine Instanz speziell für Prozentangaben.

All diese Methoden geben Ihnen ein `NumberFormat`, das Zahlen nach dem üblichen Format des Default-`Locale` verarbeitet. Sie können aber auch selbst ein `Locale`-Objekt als Parameter übergeben, um Parsing nach den Vorgaben dieser Region zu erzwingen.

In allen Fällen erhalten Sie ein Objekt vom Typ `NumberFormat`, dessen `parse`-Methode einen `String`-Parameter erwartet und eine `Number` zurückgibt. Wo möglich ist der Rückgabewert vom Typ `Long`, sonst vom Typ `Double`. Sie müssen daraus selbst den benötigten Zieltyp herstellen.

```
NumberFormat deutsch = NumberFormat.getInstance(Locale.GERMAN);
double zahl = deutsch.parse(zahlAlsString).doubleValue();
```

Listing 8.25 Eine Zahl nach deutschem Muster parsen

Sie können `NumberFormat`-Objekte auch benutzen, um Zahlen sauber formatiert auszugeben. Dazu dienen die `format`-Methoden, die auch von `MessageFormat` verwendet werden, wenn Sie eine Zahl in einer Nachricht ausgeben.

8.6 Zusammenfassung

Sie haben in diesem Kapitel viele nützliche Klassen und Methoden aus der Standardbibliothek kennengelernt. Sie haben gesehen, wie Sie mit beliebig großen Zahlen rechnen können, was Sie alles mit einem String tun können und was Sie noch mehr mit einem String tun können, wenn Sie reguläre Ausdrücke verwenden. Sie haben die umfangreichen, neuen Möglichkeiten kennengelernt, mit Datum, Zeit und Zeitzone umzugehen und wie Sie Ihre Programme mehrsprachig machen können.

Sie werden noch weitere Klassen aus der Standardbibliothek kennenlernen, aber vorher geht es im nächsten Kapitel um etwas, das Sie immer lieber vermeiden möchten: Fehler.

Auf einen Blick

Inhalt

Index

C

D

E

F

G

H

I

J

U

V

W

X

Z

Kai Günster ist Softwareentwickler, Autor eines Online-Magazins und mehrerer Fachbücher. Er ist Experte für Java-Technologien in verteilten Webanwendungen, HTML und JavaScript. Seine Projekterfahrung 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 schreibt zum Eintauchen und Mitmachen. Seine Fachbücher werden für ihre klare Sprache, ihren Unterhaltungswert und ihre kompakten, lehrreichen Beispiele geschätzt.

Kai Günster

Einführung in Java

678 Seiten, gebunden, Januar 2015
29,90 Euro, ISBN 978-3-8362-2867-1

www.rheinwerk-verlag.de/3601