Algorithmen

Algorithmen und Datenstrukturen

4., aktualisierte Auflage

Robert Sedgewick
Kevin Wayne

EXTRAS ONLINE

ALWAYS LEARNING

PEARSON

Algorithmen

Algorithmen

Algorithmen und Datenstrukturen

4., aktualisierte Auflage

Robert Sedgewick
Kevin Wayne

ALWAYS LEARNING

PEARSON

Bibliografische Information Der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.dnb.de abrufbar*.

Die Informationen in diesem Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig. Fast alle Produktbezeichnungen und weitere Stichworte und sonstige Angaben, die in diesem Buch verwendet werden, sind als eingetragene Marken geschützt. Da es nicht möglich ist, in allen Fällen zeitnah zu ermitteln, ob ein Markenschutz besteht, wird das ®-Symbol in diesem Buch nicht verwendet.

Authorized translation from the English language edition, entitled ALGORITHMS, 4th edition by Robert Sedgewick and Kevin Wayne, published by Pearson Education, Inc., publishing as Addison-Wesley, Copyright © 2011.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

GERMAN language edition published by PEARSON DEUTSCHLAND GMBH, Copyright © 2014.

10 9 8 7 6 5 4 3 2 1

16 15 14

ISBN 978-3-86894-184-5 (Buch)
ISBN 978-3-86326-758-2 (E-Book)

© 2014 by Pearson Deutschland GmbH
Lilienthalstr. 2, D-85399 Hallbergmoos/Germany
Alle Rechte vorbehalten
www.pearson.de
A part of Pearson plc worldwide

Programmleitung: Birger Peil, bpeil@pearson.de
Korrektorat: Katharina Pieper, Berlin
Übersetzung: Petra Alm, Saarbrücken
Fachlektorat: Andrea Baumann
Herstellung: Philipp Burkart, pburkart@pearson.de
Satz: Nadine Krumm, mediaService, Siegen (www.mediaservice.tv)
Coverdesign: Martin Horngacher, München
Coverillustration: Shutterstock.com
Druck und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala

Printed in Poland

Inhaltsverzeichnis

Vorwort

Dieses Buch präsentiert der wachsenden Zahl angehender Programmierer die wichtigsten Computeralgorithmen und die dafür erforderlichen grundlegenden Techniken. Es ist als Fachbuch für Informatikstudenten im zweiten Studienjahr konzipiert, die bereits über genügend Programmierkenntnisse verfügen und mit verschiedenen Computersystemen vertraut sind. Das Buch eignet sich aber auch zum Selbststudium oder als Referenz für alle, die Computersysteme oder Anwendungsprogramme entwickeln, da es Implementierungen von nützlichen Algorithmen sowie ausführliche Informationen zu Performancedaten und Clients enthält. Aufgrund der umfassenden Behandlung der Themen bietet sich dieses Buch als Einstieg in die Materie an.

Algorithmen und Datenstrukturen sind Teil des Curriculums in der Informatik, doch darüber hinaus nicht nur für Programmierer und Informatikstudenten interessant. Denn jeder, der einen Computer nutzt, wünscht sich schnellere Laufzeiten und die Lösung komplexerer Probleme. Die Algorithmen, die wir in diesem Buch betrachten, repräsentieren ein über die letzten 50 Jahre angehäuftes Wissen, das inzwischen unentbehrlich geworden ist. Aus Wissenschaft und Forschung sind diese algorithmischen Werkzeuge nicht mehr wegzudenken. Ihr Anwendungsspektrum reicht von Physik (Kollisionssimulation) über Molekularbiologie (Gensequenzierungen) und Konstruktion (Modellerstellung, Flugsimulatoren) bis zu Softwaresystemen wie Datenbanksysteme und Internetsuchmaschinen.

Bevor wir auf unser Hauptthema eingehen, entwickeln wir Datenstrukturen für Stapel, Warteschlangen und andere Low-Level-Abstraktionen, die wir in diesem Buch verwenden. Anschließend betrachten wir die grundlegenden Algorithmen zum Sortieren, Suchen, sowie zur Graphen- und Stringverarbeitung. Das letzte Kapitel bietet einen Überblick, durch den der Rest des Stoffes in diesem Buch in einen größeren Kontext eingebettet wird.

Besondere Merkmale

Dieses Buch untersucht Algorithmen, die sich in der Praxis als eminent nützlich erwiesen haben. Es beschreibt und erläutert die verschiedensten Algorithmen und Datenstrukturen so ausführlich, dass die Leser sie ohne Weiteres in jeder Systemumgebung implementieren, debuggen und ausführen können. Unser Ansatz umfasst die folgenden Bereiche:

Algorithmen

Die Beschreibungen der Algorithmen basieren auf vollständigen Implementierungen und umfassen eine Diskussion der Ausführung dieser Programme auf einer passenden Beispielmenge. Anstatt Pseudocode präsentieren wir Ihnen realen Code, sodass die Pro-

gramme schnell in der Praxis angewendet werden können. Unsere Programme sind in Java geschrieben, aber so allgemein gehalten, dass ein Großteil unseres Codes für die Entwicklung von Implementierungen in anderen modernen Programmiersprachen wiederverwendet werden kann.

Datentypen

Unser Programmierstil ist modern, das heißt, er basiert auf Datenabstraktion, sodass Algorithmen und Datenstrukturen zusammen gekapselt sind.

Anwendungen

Jedes Kapitel umfasst eine ausführliche Diskussion der Anwendungen, in denen die beschriebenen Algorithmen eine wichtige Rolle spielen. Diese reichen von Anwendungen in der Physik und Molekularbiologie über Konstruktion von Systemen bis zu vertrauten Aufgaben wie Datenkompression und Websuche.

Ein wissenschaftlicher Ansatz

Wir legen Wert darauf, mathematische Modelle zu entwickeln, die die Performance der Algorithmen beschreiben, um mithilfe dieser Modelle Hypothesen über die Performance anzustellen und dann diese Hypothesen durch Ausführen der Algorithmen in realistischen Kontexten zu prüfen.

Themenumfang

Wir behandeln in diesem Buch abstrakte Datentypen, Sortieralgorithmen, Suchalgorithmen, Graphenverarbeitung und Stringverarbeitung. Wir achten darauf, Datenstrukturen, Algorithmendesignparadigmen, Reduktion und Problemlösungsmodelle immer im Kontext der dazugehörigen Algorithmen zu betrachten. Neben den klassischen Verfahren, die es seit den 1960ern gibt, stellen wir neue Verfahren vor, die erst in den letzten Jahren entdeckt und entwickelt wurden.

Unser primäres Ziel ist es, die derzeit wichtigsten Algorithmen einem möglichst breiten Leserkreis nahezubringen. Diese Algorithmen sind im Allgemeinen geniale Entwicklungen, die sich mit erstaunlich wenig Code ausdrücken lassen – manche bestehen nur aus zwei Codezeilen. Zusammengenommen sind sie unglaublich leistungsfähig. Ihnen verdanken wir die Möglichkeit zur Programmerstellung, die Lösung komplexer wissenschaftlicher Probleme und die Entwicklung kommerzieller Anwendungen – alles Aufgaben, die ohne diese Algorithmen nicht lösbar gewesen wären.

Die Website zum Buch

Dieses Buch zeichnet sich dadurch aus, dass es durch eine Website ergänzt wird, die unter *algs4.cs.princeton.edu* frei verfügbar ist und eine Fülle an Materialien zu Algorithmen und Datenstrukturen für Dozenten, Studenten und professionelle Programmierer enthält. Sie finden dort unter anderem:

Eine Online-Zusammenfassung

Der Text auf der Website zum Buch orientiert sich vom Aufbau her an dem Buch und wird ergänzt durch Links, um Ihnen die Navigation durch das Material zu erleichtern.

Vollständige Implementierungen

Alle Codebeispiele im Buch finden Sie in aufbereiteter Form auch auf unserer Website. Darüber hinaus gibt es viele weitere Beispiele, einschließlich anspruchsvolleren Implementierungen, im Buch besprochenen Verbesserungen, Antworten auf ausgewählte Übungen und Client-Code für verschiedene Anwendungen. Unser Schwerpunkt liegt auf dem Testen der Algorithmen im Rahmen realistischer Anwendungen.

Übungen und Antworten

Sie finden auf der Website zum Buch nicht nur die Übungen zum Buch, sondern auch ein breites Spektrum an Beispielen, die die ganze Bandbreite des behandelten Stoffes veranschaulichen, Programmierübungen mit Codelösungen sowie programmiertechnische Herausforderungen.

Dynamische Visualisierungen

Dynamische Simulationen sind in der Druckausgabe eines Buches nicht möglich. Aus diesem Grund finden Sie auf der Website zum Buch eine Fülle an Implementierungen, die eine Grafik-Klasse verwenden, um ansprechende Visualisierungen zu präsentieren.

Kursunterlagen

Das Buch und die dazugehörige Website werden durch einen vollständigen Satz an Vorlesungsfolien ergänzt. Darüber hinaus gibt es eine Fülle an Programmieraufgaben mit Checklisten, Testdaten und vorbereitenden Materialien.

Verweise auf verwandtes Material

Hunderte von Links bieten den Lesern und Studenten Zugriff auf Hintergrundinformationen zu den Anwendungen sowie auf weiterführende Literatur zu den einzelnen Kapiteln und Themenschwerpunkten.

Unser Ziel bei der Zusammenstellung dieser Materialien war es, den im Buch behandelten Stoff sinnvoll zu ergänzen. Generell sollten Sie zuerst das Buch lesen, wenn Sie bestimmte Algorithmen kennenlernen wollen oder sich einen allgemeinen Überblick verschaffen möchten, und die Website zum Buch nur als Referenz für die Programmierung heranziehen beziehungsweise als Ausgangspunkt verwenden, wenn Sie online nach weiterführenden Informationen suchen.

Das Buch als Unterrichtsmittel

Dieses Buch ist als Lehrbuch für Informatikstudenten im zweiten Studienjahr konzipiert. Es deckt die wichtigsten Themen vollständig ab und eignet sich hervorragend, um Studenten einfache und fortgeschrittene Kenntnisse zur Programmierung, quantitativem Denken und zur Problemlösung zu vermitteln. In der Regel reicht es, wenn der Student das erste Jahr erfolgreich absolviert hat – der Leser sollte mindestens eine moderne Programmiersprache beherrschen und mit den Grundlagen moderner Computersysteme vertraut sein.

Die Algorithmen und Datenstrukturen sind in Java geschrieben, aber durchaus auch für diejenigen verständlich, die in einer anderen modernen Programmiersprache zu Hause sind. Wir verwenden moderne Java-Abstraktionen (einschließlich Generics), verzichten aber bewusst auf exotische Features dieser Sprache.

Die Mathematik, die Sie zur Analyse der Algorithmen und Beispiele benötigen, wird zum größten Teil im Buch erläutert (oder Sie werden darauf hingewiesen, dass genauere Ausführungen den Rahmen des Buches sprengen), sodass Sie für einen Großteil des Buches mit normalen Mathematikkenntnisse auskommen, auch wenn fortgeschrittene Kenntnisse definitiv hilfreich sind. Die Anwendungen beschäftigen sich mit grundlegenden Problemen der Wissenschaften, die ebenfalls selbsterklärend sind.

Die hier behandelten Themen gehören zum Rüstzeug aller Studenten, die ihren Abschluss in Informatik machen möchten, und sind von großem Nutzen für Studenten, die sich für Naturwissenschaften, Mathematik oder Technik interessieren.

Kontext

Dieses Buch ist die Fortsetzung unseres Werkes „Einführung in die Programmierung mit Java“, das die Grundlagen für die Programmierung in Java legt. Zusammen bilden diese beiden Bücher einen zwei- bis dreisemestrigen Einführungskurs in die Informatik, der jedem Studenten das notwendige Hintergrundwissen vermittelt, um in allen naturwissenschaftlichen, technischen oder sozialwissenschaftlichen Bereichen erfolgreich zu programmieren.

Dieses Buch basiert größtenteils auf den Vorläuferbüchern von Sedgewick zu dieser Auflage. Vom Aufbau und Inhalt ist es der ersten und zweiten Auflage dieses Buches sehr ähnlich, profitiert jedoch von den jahrzehntelangen Unterrichtserfahrungen der Autoren. Die dritte Auflage, das aktuelle Vorläuferbuch „Algorithmen in C/C++/Java, Dritte Auflage“ eignet sich dagegen eher als Referenz oder Fachbuch für einen Fortgeschrittenenkurs. Das hier vorgestellte Werk ist speziell als Fachbuch für einen einsemestrigen Kurs für Studenten im ersten oder zweiten Studienjahr gedacht sowie als moderne Einführung in die Grundlagen und als Referenz für professionelle Programmierer.

Danksagung

Dieses Buch gibt es seit fast 40 Jahren, sodass wir nicht alle, die daran mitgewirkt haben, namentlich erwähnen können. Stellvertretend für die Dutzenden von Mitwirkenden an früheren Auflagen seien hier (in alphabetischer Reihenfolge) unter anderem Andrew Appel, Trina Avery, Marc Brown, Lyn Dupré, Philippe Flajolet, Tom Freeman, Dave Hanson, Janet Incerpi, Mike Schidlowsky, Steve Summit und Chris Van Wyk genannt. Danke ihnen allen, auch wenn einige ihrer Beiträge Jahrzehnte zurückliegen. Bedanken möchten wir uns in dieser 4. Auflage aber vor allem bei den Hunderten von Studenten der Princeton University und mehrerer anderer Institute, die sich durch die Vorabversionen dieses Werkes gequält haben, sowie bei den Lesern auf der ganzen Welt, die uns über die Website zum Buch ihre Anregungen und Korrekturen zugesandt haben.

Dankbar sind wir auch für die Unterstützung durch die Princeton University, die mit ihren hohen Anforderungen an Bildung und Lehre die Grundlage für dieses Werk geschaffen hat.

Und, last not least, was wäre diese 4. Auflage ohne die exzellente Betreuung bei Pearson. Peter Gordon stand uns fast von Anfang an bei der Entstehung dieses Werkes mit seinen klugen Ratschlägen zur Seite. Ihm verdanken wir, dass in dieser Auflage der Schwerpunkt wieder mehr auf den Grundlagen liegt. Ein Dankeschön auch an Barbara Wood für ihr sorgfältiges und professionelles Lektorat, an Julie Nahil für die Produktion und an alle anderen, die im Verlag an der Erstellung und Vermarktung dieses Buches beteiligt waren. Sie alle haben sehr darauf geachtet, dass die Einhaltung des ziemlich engen Zeitrahmens nicht zulasten der Qualität ging.

Robert Sedgewick
Kevin Wayne

Princeton, NJ
Januar 2011

Vorwort zur deutschen Ausgabe

Den Autoren, Robert Sedgewick und Kevin Wayne, ist mit der 4. Auflage des Klassikers „Algorithmen“ wieder ein umfassender Überblick über die wichtigsten Datenstrukturen und Algorithmen gelungen. Die neueste Auflage enthält zusätzlich zur Aktualisierung der bisherigen Inhalte auch neue mächtige Algorithmen aus unterschiedlichen Anwendungsbereichen. Außerdem finden sich alle Datenstrukturen und Algorithmen wieder in einem Band. Die Übersetzung der 4. Auflage dieses Klassikers bietet nun auch dem deutschsprachigen Leser wieder einen Zugang zu den wichtigsten Datenstrukturen und Algorithmen.

Das erste Kapitel ist eine Einführung in das Fachgebiet Datenstrukturen und Algorithmen. Hier lernen Sie die Grundlagen zum Verständnis der folgenden Kapitel kennen. Außerdem werden hilfreiche Bibliotheken vorgestellt, die Sie später für die Lösung der im Buch gestellten Aufgaben verwenden können. Über die grundlegende Terminologien und Datenstrukturen hinaus findet auch eine erste Fallstudie anhand des Union-Find-Algorithmus statt. Lesern, die eine ausführlichere Einführung benötigen oder die Programmiersprache Java vorab kennenlernen wollen, empfehle ich das Werk „Einführung in die Programmierung mit Java“. Dieses Buch wurde ebenfalls von Robert Sedgewick und Kevin Wayne verfasst. Dort werden schon einige Konzepte und Algorithmen vorgestellt und es ist ein nahtloser Übergang in das vorliegende Buch möglich.

Kapitel zwei bis fünf enthalten unentbehrliches Basiswissen und decken ein breites Spektrum an Datenstrukturen und Algorithmen zu den Themen Sortieren, Suchen, Graphen- oder Zeichenkettenverarbeitung ab. Das letzte Kapitel stellt die Beziehung zu vielen anderen Algorithmen her, die sich an die hier vorgestellten anschließen oder diese auch ergänzen. Ein weiterer Schwerpunkt ist die Diskussion über Komplexität von Algorithmen. Hier wird z.B. gezeigt, dass schon eine kleine Änderung an der Problemstellung die Berechnung der Lösung dieses Problems in vertretbarer Zeit unmöglich macht.

Für alle vorgestellten Datenstrukturen und Algorithmen liegen im Buch Java-Implementierungen vor, die der Leser nachvollziehen und sofort verwenden kann. Besonders interessant ist der konkrete Einsatz der Algorithmen an realen Anwendungsbeispielen aus Forschung, Technik und Wirtschaft. Dabei kommt auch hier nicht Pseudocode, sondern eine funktionstüchtige Implementierung in Java zum Einsatz.

Die didaktisch hervorragend aufbereiteten Kapitel mit historischem Hintergrundwissen, wissenschaftlichen Analysen, klar strukturiertem Code, visualisierten Ablaufprotokollen und Übungen in verschiedenen Schwierigkeitsstufen eignen sich sehr gut zum Einsatz in der Lehre und für ein Selbststudium.

Webinhalte zum vorliegenden Buch

Für alle vorgestellten Datenstrukturen und Algorithmen liegen im Buch, wie schon erwähnt, die Java-Implementierungen vor, die der Leser nachimplementieren kann. Diese Implementierungen stehen aber auch online unter *http://www.cs.princeton.edu/algs4/top50* zur Verfügung.

Darüber hinaus stehen dem Leser unter dieser Adresse weitere englischsprachige Webinhalte zur Verfügung, die jedoch nicht übersetzt wurden. Die Struktur der Webseite entspricht der Kapitelstruktur des Buches. Auch der deutschsprachige Leser wird sich schnell auf dieser Webseite zurecht finden und die gewünschten Java-Implementierungen finden, da die Programmtexte des vorliegenden Buches unverändert übernommen wurden und lediglich die Kommentare übersetzt wurden. Darüber hinaus findet der Leser auf den englischsprachigen Webseiten die Testdaten, die im Buch verwendet werden. Zum Selbststudium in Englisch bietet sich auch ein Link auf die Lehrplattform *www.coursera.org* an.

Auch für Lehrende bieten die englischsprachigen Webinhalte zum Buch viele Anregungen, wie Sie Ihre Vorlesungen strukturieren oder Ihre Folien gestalten können, außerdem finden Sie hier viele Animationen der besprochenen Algorithmen. Die Folien auf den Webseiten sind ebenfalls auf Englisch. Das Onlinematerial ist sehr umfangreich, ich kann das Material aus eigener Erfahrung empfehlen.

Ich wünsche Ihnen viel Spaß beim Lesen, Lernen und Lehren.

Andrea Baumann

Grundlagen

1

ÜBERBLICK

EINLEITUNG

» Ziel dieses Buches ist das Studium und die Analyse eines breiten Spektrums an wichtigen und nützlichen *Algorithmen* – Verfahren zum Lösen von Problemen, die sich auf einem Computer implementieren lassen. Eng verbunden mit den Algorithmen sind die sogenannten *Datenstrukturen*, deren Aufgabe es ist, Daten so zu organisieren, dass sie sich effizient von den Algorithmen verarbeiten lassen. Die grundlegenden Werkzeuge, die für die Untersuchung der Algorithmen und Datenstrukturen benötigt werden, sind Thema dieses Kapitels.

Als Erstes stellen wir Ihnen unser *grundlegendes Programmiermodell* vor. Alle unsere Programme werden mithilfe einer kleinen Teilmenge der Java-Programmiersprache sowie ein paar eigenen Bibliotheken (Ein-/Ausgabe und statistische Berechnungen) erstellt. Abschnitt 1.1 gibt einen Überblick über die Sprachkonstrukte, Features und Bibliotheken, die wir in diesem Buch verwenden.

Danach wenden wir uns dem Thema *Datenabstraktion* zu, bei dem es um die Definition *abstrakter Datentypen* (ADTs) zur Unterstützung der modularen Programmierung geht. In Abschnitt 1.2 zeigen wir Ihnen, wie man abstrakte Datentypen in Java in zwei Schritten implementieren kann: durch Angabe einer API, d.h. einer Schnittstelle für die Anwendungsprogrammierung (*Applications Programming Interface*), und die Implementierung mithilfe des Java-Klassenmechanismus zur Verwendung im Client-Code.

In Abschnitt 1.3 lernen Sie dann drei überaus wichtige und nützliche Beispiele für abstrakte Datentypen kennen: *Multimenge* (*bag*), *Warteschlange* (*queue*) und *Stapel* (*stack*). Wir betrachten deren APIs und Implementierungen unter Verwendung von Arrays, Arrays variabler Größe und verketteten Listen, die im weiteren Verlauf des Buches als Modell und Ausgangsbasis für weitere Algorithmusimplementierungen dienen.

Das Laufzeitverhalten ist ein wichtiger Gesichtspunkt bei der Untersuchung und Entwickelung von Algorithmen. Abschnitt 1.4 beschreibt unseren Ansatz, die Performance eines Algorithmus zu analysieren. Dabei gehen wir von der *wissenschaftlichen Methode* aus: Wir entwickeln Hypothesen zum Laufzeitverhalten, erstellen mathematische Modelle und testen diese anhand von Experimenten – ein Prozess, den wir bei Bedarf wiederholen.

Den Abschluss dieses Kapitels bildet eine Fallstudie mit Lösungen zu einem *Zusammenhangs*problem. In der Fallstudie kommen Algorithmen und Datenstrukturen zum Einsatz, die den klassischen abstrakten *Union-Find*-Datentyp implementieren.

Algorithmen

Ein Computerprogramm zu schreiben bedeutet im Allgemeinen nichts anderes, als ein *Verfahren* zu implementieren, das zuvor dafür entwickelt wurde, ein bestimmtes Problem zu lösen. Dieses Verfahren ist meistens unabhängig von der eingesetzten Programmiersprache – mit großer Wahrscheinlichkeit dürfte es für viele Computer und viele Programmiersprachen geeignet sein. Und es ist das Verfahren und nicht das Computerprogramm, welches die Schritte vorgibt, mit denen wir das Problem lösen können. Lösungsverfahren, die endlich, deterministisch und effektiv sind und als Computerprogramm implementiert

werden können, werden in der Informatik als *Algorithmen* bezeichnet. Algorithmen sind ein wichtiges Teilgebiet der Informatik und Gegenstand zahlreicher Untersuchungen.

Man kann einen Algorithmus definieren, indem man eine Anleitung zur Lösung eines Problems in natürlicher Sprache aufsetzt oder indem man ein Computerprogramm schreibt, das diese Anleitung implementiert – wie in ▶Abbildung 1.1 am Beispiel des *euklidischen Algorithmus* demonstriert. (Der euklidische Algorithmus berechnet den größten gemeinsamen Teiler von zwei Zahlen und wurde in etwas anderer Form bereits vor 2300 Jahren formuliert. Wenn Sie mit dem euklidischen Algorithmus nicht vertraut sind, sollten Sie vielleicht schon nach dem ersten Abschnitt dieses Kapitels die *Übungen 1.1.24* und *1.1.25* durcharbeiten.) In diesem Buch beschreiben wir Algorithmen mittels Computerprogrammen. Ein wichtiger Grund dafür ist, dass wir dann leichter prüfen können, ob die Algorithmen wie gefordert endlich, deterministisch und effektiv sind. Dennoch sollte Ihnen klar sein, dass ein Programm – in welcher Sprache auch immer – nur eine Möglichkeit ist, einen Algorithmus auszudrücken. Die Tatsache, dass viele der hier beschriebenen Algorithmen im Laufe der letzten Jahrzehnte in vielen Programmiersprachen ausgedrückt worden sind, stützt unsere Behauptung, dass ein Algorithmus ein Verfahren ist, das sich zur Implementierung auf jedem Computer und in jeder Programmiersprache eignet.

Beschreibung in Deutsch

Berechne den größten gemeinsamen Teiler von zwei nicht-negativen ganzen Zahlen p und q wie folgt:

Wenn q gleich 0 ist, lautet die Antwort p. Wenn nicht, teile p durch q und nimm den Rest r. Die Antwort ist der größte gemeinsame Teiler von q und r.

Beschreibung in Java

```
public static int gcd(int p, int q)
{
   if (q == 0) return p;
   int r = p % q;
   return gcd(q, r);
}
```

Abbildung 1.1: Euklidischer Algorithmus

Die meisten interessanten Algorithmen setzen voraus, dass die Daten vor der Berechnung in irgendeiner Form organisiert werden. Eine solche Organisation führt zu *Datenstrukturen*, die ebenfalls wissenschaftlicher Untersuchungsgegenstand der Informatik sind. Algorithmen und Datenstrukturen gehören also mehr oder weniger untrennbar zusammen. In diesem Buch gehen wir davon aus, dass Datenstrukturen die Neben- oder Endprodukte von Algorithmen sind. Wir müssen sie sorgfältig studieren, um die Algorithmen zu verstehen. Einfache Algorithmen können in komplizierte Datenstrukturen resultieren und umgekehrt können komplizierte Algorithmen einfache Datenstrukturen einsetzen. Wir werden uns in diesem Buch eingehend mit den Eigenschaften vieler Datenstrukturen beschäftigen, sodass der Titel dieses Buches genauso gut auch *Algorithmen und Datenstrukturen* hätte lauten können.

Wenn wir einen Computer zur Lösung eines Problems einsetzen, gibt es normalerweise mehrere Ansätze. Bei kleinen Problemen spielt es kaum eine Rolle, wie wir vorgehen, solange das Problem korrekt gelöst wird. Bei sehr großen Problemen (oder Anwendungen, in denen wir eine riesige Anzahl kleiner Probleme lösen müssen) steigt dagegen der Druck, eigene Ansätze zu entwickeln, die Laufzeit und Speicher effizienter nutzen.

Der Hauptgrund, sich mit Algorithmen zu beschäftigen, ist, dass sich damit riesige Einsparungen erzielen lassen, ja, dass wir sogar Aufgaben in Angriff nehmen können, die ohne einen effizienten Algorithmus überhaupt nicht durchführbar wären. In Anwendungen, die Millionen von Objekten verarbeiten, ist es durchaus möglich, Programme mithilfe eines gut entworfenen Algorithmus um den Faktor eine Million und mehr zu beschleunigen. Sie werden in diesem Buch viele Beispiele hierzu sehen. Im Gegensatz dazu erreichen Sie durch Kauf und Installation eines neuen Rechners lediglich eine potenzielle Verbesserung der Programmleistung um den Faktor 10 bis 100. Ein sorgfältiger Algorithmusentwurf spielt eine extrem wichtige Rolle beim Lösen eines riesigen Problems – gleichgültig in welchem Anwendungsbereich.

Bei der Entwicklung eines sehr umfangreichen oder komplexen Computerprogramms muss viel Mühe darauf verwendet werden, das zu lösende Problem zu verstehen und zu definieren, seine Komplexität in den Griff zu bekommen und das Problem in kleinere Teilprobleme zu zerlegen, die sich leichter implementieren lassen. Oft lassen sich viele der Algorithmen, die nach der Zerlegung benötigt werden, problemlos implementieren. Daneben gibt es in den meisten Fällen aber auch Algorithmen, deren Wahl kritisch ist, da die Ausführung dieser Algorithmen einen Großteil der Systemressourcen bindet. Und genau auf diese Algorithmen wollen wir uns in diesem Buch konzentrieren. Wir werden uns mit grundlegenden Algorithmen befassen, die herangezogen werden können, um anspruchsvolle Probleme in vielen verschiedenen Anwendungsbereichen zu lösen.

Die gemeinsame Nutzung von Programmen und Code gewinnt immer größere Bedeutung, weshalb davon auszugehen ist, dass Sie von den Algorithmen in diesem Buch zwar viele *verwenden*, aber wohl nur einen kleinen Teil davon jemals selbst *implementieren* werden. So enthalten zum Beispiel die Java-Bibliotheken bereits Implementierungen einer Vielzahl von grundlegenden Algorithmen. Trotzdem ist es von Vorteil, einfache Versionen von grundlegenden Algorithmen selbst zu implementieren, da dies hilft, die Algorithmen besser zu verstehen und später anspruchsvollere Versionen aus den Bibliotheken effektiver nutzen und an die eigenen Bedürfnisse anpassen zu können. Noch wichtiger ist allerdings, dass die grundlegenden Algorithmen häufig neu implementiert werden müssen. Der Hauptgrund dafür ist, dass in viel zu kurzen Abständen vollkommen neue Computerumgebungen (Hardware und Software) mit neuen Features auf den Markt kommen, die von den alten Implementierungen nicht optimal genutzt werden. In diesem Buch konzentrieren wir uns auf die am leichtesten nachzuvollziehenden Implementierungen der besten Algorithmen. Wir schenken dabei der Codierung der kritischen Teile der Algorithmen besondere Beachtung und bemühen uns, Sie darauf hinzuweisen, wo Low-Level-Optimierungen am sinnvollsten ist.

Die Wahl des besten Algorithmus für eine bestimmte Aufgabe kann ein komplizierter Prozess sein, der unter Umständen einer anspruchsvollen mathematischen Analyse

bedarf. Der Zweig der Informatik, der sich mit diesen Fragen befasst, wird auch *Algorithmenanalyse* genannt. Für viele der Algorithmen, die wir hier vorstellen, hat die mathematische Analyse ausgezeichnete theoretische Laufzeitwerte ermittelt; von anderen Algorithmen wissen wir lediglich durch Erfahrung, dass sie gut funktionieren. Neben unserem primären Ziel, das darin besteht, Ihnen vernünftige Algorithmen für wichtige Aufgaben vorzustellen, werden wir daher immer auch großen Wert auf die vergleichende Analyse des Laufzeitverhaltens der vorgestellten Verfahren legen. Schließlich sollten Sie keinen Algorithmus verwenden, ohne eine Vorstellung davon zu haben, wie viele Ressourcen er bindet und wie effizient er arbeitet.

Zusammenfassung der Themen

Um Ihnen einen Überblick zu geben, finden Sie nachfolgend eine kurze Übersicht über die Hauptkapitel des Buches, die die darin behandelten Themen angibt und unsere Sicht auf das Material beschreibt. Bei der Auswahl der Themen haben wir sehr darauf geachtet, möglichst viele grundlegende Algorithmen abzudecken. So gehören etliche der behandelten Bereiche zu den klassischen Kernbereichen der Informatik, die wir hier ausführlich untersuchen werden, da sie elementare Algorithmen mit einer großen Anwendungsbreite enthalten. Andere der hier diskutierten Algorithmen entstammen fortgeschrittenen Feldern der Informatik oder verwandter Disziplinen. Die untersuchten Algorithmen sind das Ergebnis jahrzehntelanger Forschung und Entwicklung und werden auch in Zukunft eine wichtige Rolle in den immer anspruchsvoller werdenden Computeranwendungen spielen.

Grundlagen

(*Kapitel 1*) In diesem Buch verstehen wir darunter die grundlegenden Prinzipien und Methoden, mit denen wir Algorithmen implementieren, analysieren und vergleichen. Wir betrachten neben unserem Java-Programmiermodell noch Datenabstraktion, grundlegende Datenstrukturen, abstrakte Datentypen für Sammlungen, Methoden zur Analyse des Laufzeitverhaltens von Algorithmen und eine Fallstudie.

Sortieren

(*Kapitel 2*) Sortieralgorithmen, die die Reihenfolge der Elemente in Arrays ändern, sind von großer Bedeutung. Wir beschäftigen uns ausführlich mit einer Auswahl einschlägiger Algorithmen, einschließlich Insertionsort, Selectionsort, Shellsort, Quicksort, Mergesort und Heapsort. Wir werden aber auch auf Algorithmen für verwandte Probleme zu sprechen kommen, wie Vorrangwarteschlangen (*priority queues*), Auswahl (*selection*) und Mischen (*merging*). Viele dieser Algorithmen werden Sie später im Buch als Grundlage für andere Algorithmen wiederfinden.

Suchen

(*Kapitel 3*) Ebenfalls von großer Bedeutung sind Suchalgorithmen, um bestimmte Elemente in einer großen Sammlung von Elementen zu finden. Wir stellen grundlegende und fortgeschrittene Methoden zum Suchen vor, einschließlich binäre Suchbäume,

balancierte Suchbäume und Hashing. Wir werden feststellen, dass es Beziehungen zwischen ihnen gibt, und wir werden ihre Performance vergleichen.

Graphen

(*Kapitel 4*) Hinter Graphen verbergen sich Mengen von Objekten und Verbindungen, eventuell gewichtet und gerichtet. Graphen sind nützliche Modelle für eine Vielzahl von schwierigen und wichtigen Problemen, und der Entwurf von Algorithmen zur Verarbeitung von Graphen ist ein bedeutendes Forschungsfeld. Wir beschäftigen uns mit Tiefensuche, Breitensuche, Zusammenhangsproblemen und verschiedenen Algorithmen und Anwendungen, einschließlich der Algorithmen von Kruskal und Prim zur Ermittlung des minimalen Spannbaums und der Algorithmen von Dijkstra und Bellman-Ford zur Lösung der Probleme der kürzesten Pfade.

Strings

(*Kapitel 5*) Strings sind ein unentbehrlicher Datentyp in modernen Computeranwendungen. Wir betrachten eine Reihe von Verfahren zur Verarbeitung von Zeichenketten. Wir beginnen mit schnelleren Algorithmen zum Sortieren und Suchen, wenn die Schlüssel Strings sind. Anschließend wenden wir uns der Teilstringsuche, der musterbasierten Suche (*pattern matching*) mit regulären Ausdrücken und den Algorithmen zur Datenkomprimierung zu. Auch hier erhalten Sie eine Einführung in fortgeschrittene Themen, indem wir einige elementare Probleme behandeln, die für sich betrachtet schon wichtig und interessant sind.

Kontext

(*Kapitel 6*) Dieses Kapitel rundet das Thema ab, indem es Ihnen hilft, den Stoff dieses Buches auf andere Studiengebiete, wie wissenschaftliches Rechnen, Operations Research und die theoretische Informatik, zu übertragen. Wir geben Ihnen eine kurze Einführung in ereignisbasierte Simulation, B-Bäume, Suffixarrays, maximalen Fluss und andere fortgeschrittene Themen, die Ihnen einen Eindruck davon vermitteln sollen, in wie vielen interessanten, aktuellen Forschungsbereichen Algorithmen eine wichtige Rolle spielen. Zum Schluss gehen wir noch auf Suchprobleme, Reduktion und NP-Vollständigkeit ein, die das theoretische Fundament des Studiums der Algorithmen sind und in Beziehung zu dem Stoff dieses Buches stehen.

Das Studium der Algorithmen ist interessant und aufregend, weil es ein junges Forschungsgebiet (fast alle hier vorgestellten Algorithmen sind jünger als 50 Jahre, einige wurden gerade erst entdeckt) mit einer großen Vergangenheit ist (einige Algorithmen sind seit Hunderten von Jahren bekannt). Ständig werden neue Entdeckungen gemacht, auch wenn letztlich nur wenige Algorithmen vollständig verstanden werden. In diesem Buch betrachten wir neben komplizierten und schwierigen Algorithmen auch elegante, einfache. Die Herausforderung für uns liegt darin, Erstere zu verstehen und Letztere im Rahmen wissenschaftlicher und wirtschaftlicher Anwendungen schätzen zu lernen. Dabei werden wir eine Vielfalt an nützlichen Werkzeugen kennenlernen und eine Form des *algorithmischen Denkens* entwickeln, die uns bei zukünftigen Herausforderungen als Programmierer gute Dienste leisten wird. «

1.1 Das grundlegende Programmiermodell

Die hier betrachteten Algorithmen werden ausnahmslos als *Programme* in der Programmiersprache Java implementiert. Für diese Vorgehensweise gibt es mehrere Gründe:

- Unsere Programme sind präzise, elegante und vollständige Beschreibungen der Algorithmen.
- Sie können die Programme ausführen, um die Eigenschaften der Algorithmen zu studieren.
- Sie können die Algorithmen direkt in Ihre Anwendungen einbauen.

Dies sind wichtige und evidente Vorteile verglichen mit der Alternative, mit natürlichsprachlichen Beschreibungen der Algorithmen arbeiten zu müssen.

Ein möglicher Nachteil dieses Ansatzes ist, dass wir mit einer konkreten Programmiersprache arbeiten müssen, was es schwierig machen kann, den Entwurf eines Algorithmus von den Details seiner Implementierung zu trennen. Unsere Implementierungen zielen jedoch darauf ab, diesem Problem entgegenzuwirken, indem Programmkonstrukte zum Einsatz kommen, die in vielen modernen Programmiersprachen zu finden sind und die benötigt werden, um die Algorithmen angemessen zu beschreiben.

Wir verwenden nur einen kleinen Teil von Java, wollen hier jedoch davon absehen, diesen Teil formal zu definieren. Sicherlich werden Sie bemerken, dass nur relativ wenige Java-Konstrukte zum Einsatz kommen und wir diejenigen bevorzugen, die in vielen modernen Programmiersprachen zu finden sind. Der hier präsentierte Code ist vollständig und wir hoffen, dass Sie ihn herunterladen und auf unseren (oder eigenen) Testdaten ausführen.

Wir bezeichnen das Zusammenspiel aus Programmkonstrukten, Software-Bibliotheken und Betriebssystemfeatures, die wir zur Implementierung und Beschreibung der Algorithmen verwenden, als unser *Programmiermodell*, das wir in diesem und dem folgenden Abschnitt ausführlich beschreiben. Die Abhandlung dieses Themas ist in sich abgeschlossen und dient primär der Dokumentation und Referenzzwecken, damit Sie die einzelnen Codebeispiele in diesem Buch nachvollziehen können. Das Modell, das wir hier beschreiben, ist das gleiche, das wir in unserem Buch *Einführung in die Programmierung mit Java* vorgestellt haben – dort allerdings wesentlich ausführlicher.

▶Abbildung 1.2 zeigt ein vollständiges Java-Programm, das viele der grundlegenden Bestandteile unseres Programmmodells veranschaulicht. Wir verwenden diesen Code für Beispiele, wenn wir Sprachelemente besprechen, werden ihn jedoch erst in Abschnitt 1.1.10 im Detail betrachten (das Java-Programm in Abbildung 1.2 implementiert einen klassischen Algorithmus, die sogenannte *Binäre Suche*, und testet ihn anhand einer Anwendung, dem *Filtern mit Positivliste,* auch *Weiße Liste* (*whitelist*) genannt). Wir gehen davon aus, dass Sie Programmiererfahrungen in mindestens einer modernen Programmiersprache mitbringen, sodass Ihnen viele der Elemente in diesem Code bekannt sein sollten. Die Beschriftungen zum Code in Abbildung 1.2 enthalten Seitenverweise, damit Sie leichter Antworten zu eventuellen Fragen finden. Da unser Code etwas stili-

siert ist, wir uns aber bemühen, die verschiedenen Java-Idiome und -Konstrukte konsistent zu verwenden, lohnt es sich auch für erfahrene Java-Programmierer die Hinweise in diesem Abschnitt zu lesen.

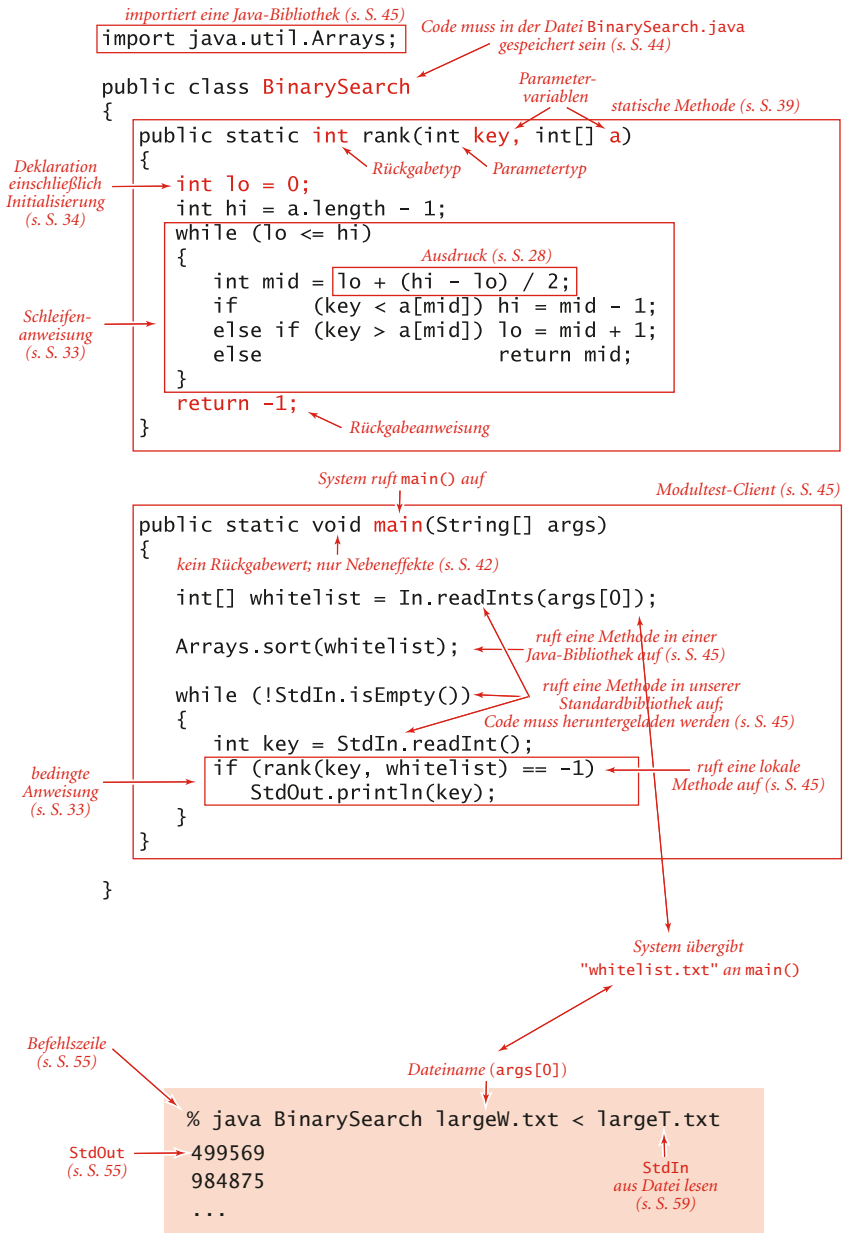


Abbildung 1.2: Aufbau eines Java-Programms und dessen Aufruf von der Befehlszeile

1.1.1 Grundlegende Struktur eines Java-Programms

Ein Java-Programm (*Klasse*) ist entweder eine *Bibliothek von statischen Methoden* (Funktionen) oder eine *Datentypdefinition*. Um Bibliotheken von statischen Methoden und Datentypdefinitionen zu erzeugen, verwenden wir die folgenden Komponenten – die die Basis der Programmierung in Java und vielen anderen modernen Programmiersprachen bilden:

- *Primitive Datentypen* legen in einem Computerprogramm die genaue Bedeutung von Begriffen wie *Integer* (ganze Zahlen), *reelle Zahlen* und *boolesche Werte* fest. Ihre Definition umfasst den jeweiligen Bereich möglicher Werte und die *Operationen*, die auf den Werten ausgeführt werden können. Werte und Operationen können zu *Ausdrücken* zusammengefasst werden, wie mathematische Ausdrücke, die Werte definieren.
- *Anweisungen* erlauben uns, eine Berechnung zu definieren, indem wir Variablen erzeugen und ihnen Werte zuweisen, den Programmablauf steuern oder Nebeneffekte verursachen. Wir verwenden sechs Arten von Anweisungen: *Deklarationen*, *Zuweisungen*, *bedingte Anweisungen*, *Schleifen*, *Aufrufe* und *Rückgabeanweisungen*.
- *Arrays* erlauben uns, mit mehreren Werten des gleichen Typs zu arbeiten.
- *Statische Methoden* erlauben uns, Code zu kapseln und wiederzuverwenden und Programme als einen Satz von unabhängigen Modulen zu entwickeln.
- *Strings* sind Folgen von Zeichen. Einige Operationen für Strings sind in Java bereits integriert.
- *Ein-/Ausgabe* sorgt für die Kommunikation zwischen Programmen und der Außenwelt.
- *Datenabstraktion* erweitert das Prinzip der Kapselung und Wiederverwendung, sodass wir die Möglichkeit haben, nicht primitive Datentypen zu definieren, und unterstützt damit die objektorientierte Programmierung.

In diesem Abschnitt werden wir die ersten sechs Punkte der Reihe nach betrachten. Im nächsten Abschnitt werden wir uns dann dem Thema Datenabstraktion zuwenden.

Das Ausführen eines Java-Programms setzt die Interaktion mit einem Betriebssystem oder einer Programmentwicklungsumgebung voraus. Um uns hier nicht unnötig in systemspezifische Details zu verstricken, beschreiben wir solche Aktionen mithilfe eines *virtuellen Terminals*, welches uns erlaubt, mit Programmen zu interagieren, indem wir Befehle eintippen und an das System schicken. Auf der Website zum Buch finden Sie nähere Angaben, wie Sie ein virtuelles Terminal auf Ihrem System einrichten oder eine der vielen umfangreichen Programmentwicklungsumgebungen verwenden, die auf modernen Systemen zur Verfügung stehen.

Unser Beispielprogramm `BinarySearch` besteht aus zwei statischen Methoden: `rank()` und `main()`. Die erste statische Methode, `rank()`, weist vier Anweisungen auf: zwei Deklarationen, eine Schleife (die selbst aus einer Zuweisung und zwei bedingten

Anweisungen besteht) und eine Rückgabeanweisung. Die zweite, `main()`, weist drei Anweisungen auf: eine Deklaration, einen Aufruf und eine Schleife (die selbst aus einer Zuweisung und einer bedingten Anweisung besteht).

Um ein Java-Programm aufzurufen, *kompilieren* wir es zuerst mit dem Befehl `javac` und führen es dann mit dem Befehl `java` aus. Für die Ausführung von `BinarySearch` geben wir beispielsweise zuerst den Befehl `javac BinarySearch.java` ein (der eine Datei namens *BinarySearch.class* erzeugt, die eine maschinennähere Version des Programms in *Bytecode* enthält). Dann geben wir `java BinarySearch` ein (gefolgt von einem Dateinamen für eine Positivliste), um die Kontrolle an die Bytecode-Version des Programms zu übergeben. Um die Auswirkungen dieser Aktionen besser zu verstehen, wollen wir uns als Nächstes ausführlich mit primitiven Datentypen und Ausdrücken befassen, gefolgt von den verschiedenen Arten von Java-Anweisungen, Arrays, statischen Methoden, Strings sowie der Ein- und Ausgabe.

1.1.2 Primitive Datentypen und Ausdrücke

Ein *Datentyp* besteht aus einem Satz von Werten und einem Satz von Operationen auf diesen Werten. Beginnen wollen wir mit den folgenden vier *primitiven* Datentypen, die die Grundlage der Programmiersprache Java bilden:

- *Integer* (ganze Zahlen), mit arithmetischen Operationen (`int`)
- *Reelle Zahlen*, ebenfalls mit arithmetischen Operationen (`double`)
- *Boolesche Werte*, die beiden Werte `true` und `false` mit ihren logischen Operationen (`boolean`)
- *Zeichen*, die alphanumerischen Zeichen und Symbole, die Sie eingeben (`char`)

Als Nächstes betrachten wir Mechanismen, mit denen wir Werte und Operationen für diese Typen angeben.

Ein Java-Programm manipuliert *Variablen*, die durch *Bezeichner* eindeutig benannt sind. Jede Variable wird durch einen Datentyp definiert und speichert einen der erlaubten Datentypwerte. In Java-Code verwenden wir *Ausdrücke* wie vertraute mathematische Ausdrücke und wenden darauf die Operationen an, die den jeweiligen Datentypen zugeordnet sind. Bei primitiven Typen verwenden wir Bezeichner, um auf Variablen Bezug zu nehmen, *Operator*symbole wie `+`, `-`, `*` und `/` für Operationen, *Literale* wie `1` oder `3.14` zur Angabe von Werten und Ausdrücke wie `(x + 2.236)/2`, um Operationen auf Werte auszuführen. Der Zweck eines Ausdrucks ist es, einen der Datentypwerte zu definieren.

Tabelle 1.1 Grundlegende Bestandteile für Java-Programme

Begriff	Beispiel	Definition
Primitiver Datentyp	`int double boolean char`	Ein Satz von Werten und ein Satz von Operationen auf diesen Werten (integriert in Java).
Bezeichner	`a abc Ab$ a_b ab123 lo hi`	Eine Folge von Buchstaben, Ziffern, _ und $, wobei an erster Stelle keine Ziffer steht.
Variable	[*jeder Bezeichner*]	Benennt einen Datentypwert.
Operator	`+ - * /`	Benennt eine Datentypoperation.

Literal	`int` `1 0 -42`
	`double` `2.0 1.0e-15 3.14`
	`boolean` `true false`
	`char` `'a' '+' '9' '\n'`
		Quellcoderepräsentation eines Wertes.

Ausdruck	`int` `lo + (hi - lo)/2`
	`double` `1.0e-15 * t`
	`boolean` `lo <= hi`
		Ein Literal, eine Variable oder eine Folge von Operationen auf Literalen und/oder Variablen, die einen Wert ergeben.

Um einen Datentyp zu definieren, müssen wir nur die Werte angeben und den Satz von Operationen auf diesen Werten. Für die Java-Datentypen `int`, `double`, `boolean` und `char` sind diese Informationen in ▶Tabelle 1.2 zusammengefasst. Diese Datentypen entsprechen den grundlegenden Datentypen, wie sie in vielen Programmiersprachen zu finden sind. Für `int` und `double` sind die Operationen die allseits bekannten arithmetischen Operationen, für `boolean` die bekannten logischen Operationen. In diesem Zusammenhang soll darauf hingewiesen werden, dass `+`, `-`, `*` und `/` *überladen* sind, d.h., das gleiche Symbol steht je nach Kontext für Operationen verschiedener Datentypen. Die Haupteigenschaft dieser primitiven Operationen ist, *dass Operationen auf Werten eines gegebenen Datentyps einen Wert dieses Datentyps zurückliefern*. Diese Regel unterstreicht, dass wir oft mit Näherungswerten arbeiten, denn es kommt häufig vor, dass der exakte Wert, der durch den Ausdruck definiert zu werden scheint, kein Wert des Datentyps ist. So hat zum Beispiel `5/3` den Wert `1` und `5.0/3.0` einen Wert von ziemlich genau `1.66666666666667`. Aber keiner dieser Werte ist genau gleich `5/3`. Tabelle 1.2 ist selbstverständlich nicht vollständig. Wir werden weitere Operatoren und einige gelegentlich zu berücksichtigende Ausnahmesituationen in den Fragen und Antworten am Ende dieses Abschnitts vorstellen.

Tabelle 1.2 Primitive Datentypen in Java

Datentyp	Wertebereich	Operatoren	Typische Ausdrücke	
			Ausdruck	Wert

`int`	Integer zwischen -2^{31} und $+2^{31}-1$ (32-Bit- Zweierkomplement)	+ (Addition)
		- (Subtraktion)
		* (Multiplikation)
		/ (Division)
		% (Modulo)
			`5 + 3`
			`5 - 3`
			`5 * 3`
			`5 / 3`
			`5 % 3`
				`8`
				`2`
				`15`
				`1`
				`2`

`double`	Gleitkommazahlen doppelter Präzision (64-Bit-IEEE-Standard 754)	+ (Addition)
		- (Subtraktion)
		* (Multiplikation)
		/ (Division)
			`3.111 + .03`
			`2.0 - 2.0e-7`
			`100 * .015`
			`6.02e23 / 2.0`
				`3.141`
				`1.9999998`
				`1.5`
				`3.01e23`

`boolean`	`true` oder `false`	&& (Und)
		|| (Oder)
		! (Nicht)
		^ (Xor)
			`true && false`
			`false || true`
			`! false`
			`true ^ true`
				`false`
				`true`
				`true`
				`false`

`char`	Zeichen (16-Bit)	[*arithmetische Operationen, selten verwendet*]		

Ausdrücke

Wie in Tabelle 1.2 ersichtlich, werden typische Ausdrücke in *Infixnotation* geschrieben: ein Literal (oder ein Ausdruck), gefolgt von einem Operator, gefolgt von einem weiteren Literal (oder einem weiteren Ausdruck). Wenn ein Ausdruck mehr als einen Operator enthält, spielt die Reihenfolge ihrer Ausführung oft eine wichtige Rolle, weshalb die folgenden Prioritätskonventionen Teil der Java-Spezifikation sind: Die Operatoren `*` und `/` (und `%`) haben eine höhere Priorität als die Operatoren `+` und `-` (d.h., sie werden zuerst ausgeführt). Bei den logischen Operatoren hat `!` die höchste Priorität, gefolgt von `&&` und `||`. Im Allgemeinen werden Operatoren der gleichen Priorität von links nach rechts ausgeführt. Wie in normalen arithmetischen Ausdrücken können Sie Klammern verwenden, um diese Regeln außer Kraft zu setzen. Da sich die Prioritätsregeln von Programmiersprache zu Programmiersprache leicht unterscheiden, versuchen wir unter anderem durch Setzen von Klammern, die Abhängigkeit von diesen Prioritätsregeln in unserem Code zu vermeiden.

Typumwandlung

Zahlen werden automatisch in einen Datentyp mit einem größeren Wertebereich umgewandelt, wenn dabei keine Informationen verloren gehen. Diese implizite Typumwandlung wird als *Promotion* bezeichnet. Zum Beispiel wird in dem Ausdruck `1 + 2.5` der Wert `1` in den `double`-Wert `1.0` umgewandelt und die Auswertung dieses Ausdrucks liefert den `double`-Wert `3.5` zurück. Darüber hinaus gibt es auch eine explizite Typumwandlung, einen sogenannten *Cast*. Dabei setzen Sie in einem Ausdruck die Daten-

typbezeichnung in Klammern und weisen damit den Computer an, den nachfolgenden Wert in einen Wert des angegebenen Datentyps umzuwandeln. So ist beispielsweise `(int) 3.7` gleich `3` und `(double) 3` gleich `3.0`. Beachten Sie, dass bei einem Cast in einen Integer der Nachkommaanteil abgeschnitten und nicht gerundet wird – Regeln für explizite Typumwandlungen in komplexen Ausdrücken können kompliziert sein, weshalb Casts im Allgemeinen möglichst wenig und mit Vorsicht verwendet werden sollten. Am besten beschränken Sie sich in Ihren Ausdrücke auf Literale oder Variablen nur eines Datentyps.

Vergleiche

Die folgenden Operatoren vergleichen zwei Werte des gleichen Datentyps und liefern einen booleschen Wert zurück: *gleich* (`==`), *nicht gleich* (`!=`), *kleiner als* (`<`), *kleiner gleich* (`<=`), *größer als* (`>`) und *größer gleich* (`>=`). Diese Operatoren werden auch als *Vergleichs- oder Mischtypoperatoren* bezeichnet, da das Ergebnis vom Typ `boolean` ist und nicht vom Typ der verglichenen Werte. Ein Ausdruck mit einem booleschen Wert heißt auch *boolescher Ausdruck*. Wie wir bald sehen werden, sind solche Ausdrücke wesentlicher Bestandteil von bedingten Anweisungen und Schleifen.

Andere primitive Datentypen

Der Java-Datentyp `int` hat per definitionem 2^{32} verschiedene Werte, sodass er durch ein 32-Bit-Maschinenwort repräsentiert werden kann (viele Rechner haben heutzutage 64-Bit-Wörter, aber den 32-Bit-Datentyp `int` gibt es weiterhin). Entsprechend spezifiziert der `double`-Standard eine 64-Bit-Repräsentation. Diese Datentypgrößen sind für typische Anwendungen, die Integer und reelle Zahlen verwenden, durchaus ausreichend. Für mehr Flexibilität stellt Java Ihnen fünf weitere primitive Datentypen zur Verfügung:

- 64-Bit-Integer, mit arithmetischen Operationen (`long`)
- 16-Bit-Integer, mit arithmetischen Operationen (`short`)
- 16-Bit-Zeichen, mit arithmetischen Operationen (`char`)
- 8-Bit-Integer, mit arithmetischen Operationen (`byte`)
- 32-Bit-Gleitkommazahlen mit einfacher Präzision, ebenfalls mit arithmetischen Operationen (`float`)

Wir verwenden in diesem Buch hauptsächlich die arithmetischen Operationen für `int` und `double`, sodass wir hier nicht näher auf die anderen Datentypen eingehen wollen (die übrigens sehr ähnlich sind).

1.1.3 Anweisungen

Java-Programme bestehen aus *Anweisungen*, welche die eigentliche Berechnung definieren, indem sie Variablen erzeugen und manipulieren, den Variablen Datentypwerte zuweisen und den Ablauf dieser Operationen steuern. Anweisungen werden oft zu Blöcken zusammengefasst, d.h. als Anweisungsfolgen in geschweiften Klammern gesetzt.

- *Deklarationen* erzeugen Variablen eines bestimmten Datentyps und verbinden sie mit einem Bezeichner.
- *Zuweisungen* verbinden einen Datentypwert (definiert durch einen Ausdruck) mit einer Variablen. Java kennt darüber hinaus einige *implizite Zuweisungsidiome*, die den Wert eines Datentypwertes relativ zu seinem aktuellen Wert ändern (z.B. den Wert einer Integervariablen inkrementieren).
- *Bedingte Anweisungen* ändern auf einfache Art und Weise den Programmablauf. Es werden in Abhängigkeit von einer spezifizierten Bedingung die Anweisungen in einem von zwei Anweisungsblöcken ausgeführt.
- *Schleifen* unterstützen eine grundlegende Anpassung des Programmablaufs. Die Anweisungen in einem Block werden so lange ausgeführt, wie eine gegebene Bedingung `true` ist.
- *Aufrufe* und *Rückgabewerte* beziehen sich auf statische Methoden (siehe Abschnitt 1.1.6). Damit kann auf andere Weise der Fluss der Ausführung geändert und Code organisiert werden.

Ein Programm ist eine Folge von Anweisungen mit Deklarationen, Zuweisungen, bedingten Anweisungen, Schleifen, Aufrufen und Rückgabewerten. Programme weisen in der Regel eine *verschachtelte* Struktur auf: eine Anweisung innerhalb eines Blocks einer bedingten Anweisung oder einer Schleife kann selbst eine bedingte Anweisung oder eine Schleife sein. So enthält die `while`-Schleife in `rank()` eine `if`-Anweisung. Als Nächstes betrachten wir diese Anweisungstypen im Einzelnen.

Deklarationen

Eine *Deklaration* verbindet einen Variablennamen zur Kompilierzeit mit einem Typ. Java zwingt uns, die Namen und Datentypen von Variablen mittels Deklarationen zu spezifizieren. Auf diese Weise liefern wir explizite Hintergrundinformationen zu den Berechnungen, die wir formulieren. Java wird als *streng typisierte* Programmiersprache bezeichnet, weil der Java-Compiler genau auf Konsistenz prüft (beispielsweise erlaubt er nicht, einen booleschen Wert mit einem `double`-Wert zu multiplizieren). Variablen können überall vor ihrer ersten Verwendung deklariert werden – meistens steht die Deklaration *direkt* vor ihrer ersten Verwendung. Der *Gültigkeitsbereich* einer Variablen ist der Teil des Programms, in dem sie definiert ist. Im Allgemeinen besteht der Gültigkeitsbereich einer Variablen aus den Anweisungen, die auf die Deklaration folgen und im selben Block wie diese stehen.

Zuweisungen

Eine *Zuweisung* verbindet einen Datentypwert (als Ausdruck definiert) mit einer Variablen. Wenn wir in Java `c = a + b` schreiben, hat dies nichts mit mathematischer Gleichheit zu tun, sondern wir drücken vielmehr eine Aktion aus: Setze den Wert der Variablen `c` auf den Wert von `a` plus den Wert von `b`. Es stimmt zwar, dass `c` direkt

nach Ausführung der Zuweisung mathematisch gesehen gleich `a + b` ist, aber Ziel der Anweisung ist es, den Wert von `c` (falls notwendig) zu ändern. Die linke Seite einer Zuweisung muss deshalb aus einer einzigen Variablen bestehen; die rechte Seite kann ein beliebiger Ausdruck sein, der Werte dieses Typs erzeugt.

Bedingte Anweisungen

Bei den meisten Berechnungen müssen abhängig von der Eingabe unterschiedliche Verarbeitungsschritte ausgeführt werden. Eine Möglichkeit, dies in Java auszudrücken, ist die `if`-Anweisung:

```
if (<boolescher Ausdruck>) { <Blockanweisungen> }
```

Diese Beschreibung ist eine formale Notation – auch *Template* oder *Vorlage* genannt –, mit der wir gelegentlich den Aufbau von Java-Konstrukten angeben. Wir setzen ein Konstrukt, das wir bereits definiert haben, in spitze Klammern (`< >`) um anzuzeigen, dass wir überall dort, wo dies angegeben ist, eine beliebige Instanz dieses Konstrukts verwenden können. In diesem Fall steht ‹*boolescher Ausdruck*› für einen Ausdruck, der einen booleschen Wert zurückliefert, da er zum Beispiel eine Vergleichsoperation beinhaltet, und ‹*Blockanweisungen*› steht für eine Folge von Java-Anweisungen. Es wäre durchaus möglich, ‹*boolescher Ausdruck*› und ‹*Blockanweisungen*› formal zu definieren, aber wir wollen hier nicht allzu sehr ins Detail gehen. Die Bedeutung der `if`-Anweisung ist nahezu selbsterklärend: Die Anweisung(en) im Anweisungsblock sind genau dann auszuführen, wenn der Ausdruck `true` ergibt. Die folgende `if-else`-Anweisung

```
if (<boolescher Ausdruck>) { <Blockanweisungen> }
else                       { <Blockanweisungen> }
```

erlaubt es, zwischen zwei alternativen Anweisungsblöcken zu wählen.

Schleifen

Viele Berechnungen sind inhärent iterativ. Das grundlegende Java-Konstrukt für solche Berechnungen hat das folgende Format:

```
while (<boolescher Ausdruck>) { <Anweisungen> }
```

Die `while`-Anweisung hat die gleiche Form wie die `if`-Anweisung (der einzige Unterschied besteht in der Verwendung des Schlüsselwortes `while` anstelle von `if`), aber die Bedeutung ist eine ganz andere. Der Computer wird angewiesen, sich wie folgt zu verhalten: Wenn der Ausdruck `false` ist, mache nichts; wenn der Ausdruck `true` ist, führe die Folge der Anweisungen (wie bei `if`) aus, prüfe dann aber nochmals den Ausdruck und führe die Folge der Anweisungen erneut aus, wenn der Ausdruck weiterhin `true` ist. *Wiederhole* diesen Schritt so oft, wie der Ausdruck `true` ergibt. Wir bezeichnen den Anweisungsblock in einer Schleife auch als *Rumpf* der Schleife.

Break- und Continue-Anweisungen

Manche Situationen verlangen eine etwas kompliziertere Ablaufsteuerung, die mit einfachen `if`- und `while`-Anweisungen nur schwer erzeugt werden kann. Deshalb unterstützt Java zwei weitere Anweisungen, die in `while`-Schleifen Verwendung finden:

- die `break`-Anweisung, die die Schleife direkt verlässt
- die `continue`-Anweisung, die sofort mit dem nächsten Durchlauf (Iteration) der Schleife beginnt

Wir werden diese Anweisungen in diesem Buch weitestgehend vermeiden (manche Programmierer verwenden sie nie), aber in bestimmten Fällen vereinfachen sie den Code erheblich.

1.1.4 Kurzschreibweisen

Es gibt mehrere Möglichkeiten, eine gegebene Berechnung auszudrücken, wobei wir uns stets um klaren, eleganten und effizienten Code bemühen. Ein solcher Code macht oft Gebrauch von den folgenden weit verbreiteten Kurzformen (die in vielen Sprachen, nicht nur in Java zu finden sind).

Deklarationen mit Initialisierung

Wir können eine Deklaration mit einer Zuweisung kombinieren, um eine Variable bei ihrer Deklaration (Erzeugung) zu initialisieren. Zum Beispiel erzeugt der Code `int i = 1` eine Variable vom Typ `int` und weist ihr den Anfangswert `1` zu. Am besten bedienen Sie sich dieses Mechanismus möglichst kurz, bevor Sie die Variable das erste Mal verwenden (um den Gültigkeitsbereich zu beschränken).

Implizite Zuweisungen

Die folgenden Kurzformen stehen Ihnen zur Verfügung, wenn Sie den Wert einer Variablen relativ zu ihrem aktuellen Wert ändern wollen:

- Inkrement-/Dekrement-Operatoren: `i++` ist identisch mit `i = i + 1` und hat in einem Ausdruck den Wert `i`. Entsprechend ist `i--` identisch mit `i = i - 1`. Der Code `++i` und `--i` ist ähnlich, nur dass der Wert des Ausdrucks *nach* dem Inkrementieren/Dekrementieren zurückgeliefert wird und nicht vorher.
- Weitere zusammengesetzte Operationen: Wenn Sie in einer Zuweisung einen binären Operator vor das Gleichheitszeichen (=) stellen, ist der Code der gleiche, wie wenn Sie die Variable zur Linken als ersten Operanden verwenden. So ist zum Beispiel der Code `i/=2;` gleich dem Code `i = i/2;`. Beachten Sie, dass `i += 1;` das gleiche Ergebnis liefert wie `i = i+1;` (und `i++`).

Blöcke mit nur einer Anweisung

Wenn ein Anweisungsblock in einer bedingten Anweisung oder einer Schleife nur aus einer Anweisung besteht, können die geschweiften Klammern weggelassen werden.

For-Notation

Viele Schleifen weisen folgendes Muster auf: Sie initialisieren eine Indexvariable mit einem Wert und bedienen sich dann einer `while`-Schleife, um eine Schleifenabbruchbedingung unter Verwendung einer Indexvariablen zu testen, wobei die letzte Anweisung in der `while`-Schleife die Indexvariable inkrementiert. Solche Schleifen können Sie in Java auch direkt mit der `for`-Notation ausdrücken:

```
for (<Initialisierung>; <boolescher Ausdruck>; <Inkrement>)
{
   <Blockanweisungen>
}
```

Dieser Code ist bis auf einige wenige Ausnahmen identisch mit

```
<Initialisierung>
while (<boolescher Ausdruck>)
{
   <Blockanweisungen>
   <Inkrement>;
}
```

Wir verwenden `for`-Schleifen, um dieses typische „Initialisieren-und-Inkrementieren"-Idiom zu unterstützen.

Tabelle 1.3 Java-Anweisungen

Anweisung	Beispiele	Definition
Deklaration	`int i;` `double c;`	Erzeugt eine Variable eines spezifizierten Datentyps mit einem gegebenen Bezeichner als Namen.
Zuweisung	`a = b + 3;` `discriminant = b*b - 4.0*c;`	Weist einer Variablen einen Datentypwert zu.
Deklaration mit Initialisierung	`int i = 1;` `double c = 3.141592625;`	Weist bei einer Deklaration der Variablen gleich einen Anfangswert zu.
Implizite Zuweisung	`i++;` `i += 1;`	`i = i + 1;`
Bedingte Anweisung (`if`)	`if (x < 0) x = -x;`	Führt in Abhängigkeit von einem booleschen Ausdruck eine Anweisung aus.
Bedingte Anweisung (`if-else`)	`if (x > y) max = x;` `else max = y;`	Führt in Abhängigkeit von einem booleschen Ausdruck die eine oder andere Anweisung aus.

Tabelle 1.3 Java-Anweisungen *(Forts.)*

Anweisung	Beispiele	Definition

Schleife (`while`)	`int v = 0;`
	`while (v <= N)`
	`  v = 2*v;`
	
	`double t = c;`
	`while (Math.abs(t - c/t) > 1e-15*t)`
	`  t = (c/t + t) / 2.0;`
		Führt eine Anweisung aus, bis ein boolescher Ausdruck zu `false` ausgewertet wird.

Schleife (`for`)	`for (int i = 1; i <= N; i++)`
	`  sum += 1.0/i;`
	
	`for (int i = 0; i <= N; i++)`
	`  StdOut.println(2*Math.PI*i/N);`
		Kompakte Version der `while`-Anweisung.

Aufruf	`int key = StdIn.readInt();`	Ruft andere Methoden auf (siehe Abschnitt *Statische Methoden aufrufen* auf Seite 40).
Rückgabe-anweisung	`return false`	Kehrt aus einer Methode zurück (siehe Abschnitt *Eigenschaften von Methoden* auf Seite 42).

1.1.5 Arrays

Ein *Array (Feld)* speichert eine Folge von Werten, die alle den gleichen Datentyp haben. Doch wir wollen nicht nur Werte speichern, sondern auch auf jeden einzelnen Wert direkt zugreifen. Die Technik, mit der wir auf einzelne Werte in einem Array Bezug nehmen, basiert darauf, dass wir die Werte zuerst durchnummerieren und sie dann *indizieren*. Wenn wir N Werte haben, stellen wir uns vor, sie wären von 0 bis $N-1$ durchnummeriert. Dann können wir in unserem Java-Code eindeutig einen davon angeben, indem wir uns mit der Notation `a[i]` auf den i-ten Wert beziehen für jeden Wert i von `0` bis `N-1`. Das so beschriebene Java-Konstrukt wird als *eindimensionales Array* bezeichnet.

Ein Array erzeugen und initialisieren

Um in einem Java-Programm ein Array anzulegen, sind drei Schritte erforderlich:

- Das Array mit seinem Namen und Typangabe deklarieren.
- Das Array erzeugen.
- Die Arraywerte initialisieren.

Für die Deklaration des Arrays müssen Sie einen Namen für das Array und den Typ der im Array enthaltenen Daten angeben. Um es zu erzeugen, müssen Sie seine Größe (die Anzahl der Werte) angeben. Zum Beispiel erzeugen Sie mit der langen Codeversion aus ▶Abbildung 1.3 ein Array von `N` Zahlen vom Typ `double`, die alle mit dem Wert

0.0 initialisiert sind. Die erste Anweisung ist die Arraydeklaration. Abgesehen von den eckigen Klammern, die auf den Typnamen folgen und darauf hinweisen, dass wir ein Array deklarieren, unterscheidet sie sich kaum von der Deklaration einer Variablen des entsprechenden primitiven Typs. Das Schlüsselwort new in der zweiten Anweisung ist eine Java-Direktive, die das Array erzeugt. Der Grund, warum wir Arrays explizit zur Laufzeit erzeugen müssen, ist, dass der Java-Compiler nicht wissen kann, wie viel Speicherplatz er für das Array zur Kompilierzeit reservieren muss. (Diese Aktion ist für Variablen eines primitiven Typs nicht erforderlich.) Die for-Anweisung initialisiert die N Arraywerte. Der Code dieser Schleife setzt alle Arrayeinträge auf den Wert 0.0. Wenn Sie in Ihrem Code Arrays verwenden, müssen Sie sicherstellen, dass Ihr Code die Arrays deklariert, erzeugt und initialisiert. Gerade Programmieranfänger vergessen schnell den einen oder anderen dieser Schritte.

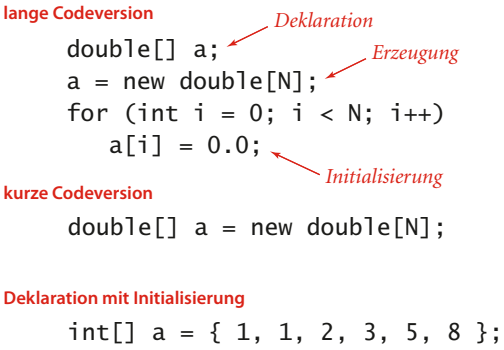


Abbildung 1.3: Deklaration, Erzeugung und Initialisierung eines Arrays

Kurze Codeversion

Um den Code möglichst kurz und prägnant zu halten, machen wir oft Gebrauch von Javas Konvention der Standard-Arrayinitialisierung und kombinieren alle drei Schritte in einer einzigen Anweisung, wie in der kurzen Codeversion in unserem Beispiel. Der Code links des Gleichheitszeichens entspricht der Deklaration und der Code rechts davon der Erzeugung des Arrays. Die for-Schleife ist in diesem Fall nicht erforderlich, da der Anfangswert von double-Variablen in einem Java-Array standardmäßig 0.0 ist. Sie wäre jedoch notwendig, wenn ein Wert ungleich null gewünscht wäre. Der Standard-Initialisierungswert für Zahlen ist null und für boolesche Werte false. Die dritte Option in unserem Beispiel besteht darin, die Initialisierungswerte zur Kompilierzeit anzugeben, und zwar in Form von durch Kommata getrennte Literalwerte in geschweiften Klammern.

Ein Array verwenden

Typischen Code zur Verarbeitung von Arrays finden Sie in ▶Tabelle 1.4. Nachdem Sie ein Array deklariert und erzeugt haben, können Sie jeden einzelnen Wert ansprechen – und zwar überall im Programm, wo Sie auch einen Variablennamen verwenden würden. Für den Zugriff geben Sie nach dem Arraynamen einen Integerindex in eckigen Klammern an. Wenn wir ein Array erzeugen, wird seine Größe festgelegt. Ein Pro-

gramm kann mit dem Code `a.length` die Länge eines Arrays `a[]` abfragen. Das letzte Element eines Arrays `a[]` ist immer `a[a.length-1]`. Java führt eine *automatische Bereichsprüfung* durch – wenn Sie ein Array der Größe `N` erzeugt haben und einen Index mit einem Wert kleiner `0` oder größer `N-1` ist, wird Ihr Programm bei der Ausführung mit einer Ausnahme vom Typ `ArrayIndexOutOfBounds` abgebrochen.

Aliasing

Denken Sie immer daran, *dass sich ein Arrayname auf das ganze Array bezieht* – wenn wir einen Arraynamen einem anderen Array zuweisen, verweisen beide auf dasselbe Array, wie das folgende Codefragment zeigt:

```
int[] a = new int[N];
...
a[i] = 1234;
...
int[] b = a;
...
b[i] = 5678;  // a[i] ist jetzt 5678.
```

Dieser Fall wird als *Aliasing* bezeichnet und kann zu schwer auffindbaren Fehlern führen. Wenn Sie beabsichtigen, eine Kopie eines Arrays zu erzeugen, müssen Sie ein neues Array deklarieren, erzeugen und initialisieren und dann alle Einträge im ursprünglichen Array in das neue Array kopieren (siehe hierzu auch das Beispiel in der dritten Zeile von Tabelle 1.4).

Zweidimensionale Arrays

Ein *zweidimensionales Array* in Java ist ein Array von eindimensionalen Arrays. Weisen die eindimensionalen Arrays verschiedene Längen auf, bezeichnen wir das zweidimensionale Array als *ausgefranst* (*ragged*). Am häufigsten arbeiten wir jedoch mit zweidimensionalen *M*×*N*-Arrays, d.h. Arrays mit *M Zeilen*, von denen jede ein Array der Länge *N* ist (sodass es durchaus sinnvoll ist, davon zu sprechen, dass das Array *N Spalten* hat). Die in Java verwendeten Arraykonstrukte können auf einfache (und naheliegende) Weise auf die zweite Dimension ausgedehnt werden. Um auf das Element in Zeile `i` und Spalte `j` eines zweidimensionalen Arrays `a[][]` zuzugreifen, verwenden wir die Notation `a[i][j]`. Um ein zweidimensionales Array zu deklarieren, fügen wir ein weiteres Paar eckige Klammern hinzu und erzeugen das Array, indem wir nach dem Typnamen die Anzahl der Zeilen gefolgt von der Anzahl der Spalten (beide in eckigen Klammern) angeben:

```
double[][] a = new double[M][N];
```

Wir bezeichnen ein solches Array als ein *M*×*N*-Array. Per Konvention ist die erste Dimension die Anzahl der Zeilen und die zweite die Anzahl der Spalten. Wie bei eindimensionalen Arrays initialisiert Java alle Einträge in Arrays von Zahlen mit null und in Arrays von booleschen Werten mit `false`. Die Standardinitialisierung für zweidimensionale Arrays ist besonders nützlich, da sie mehr Code maskiert als für eindimensionale Arrays. Der folgende Code entspricht dem einzeiligen Erzeugen-und-Initialisieren-Code, den wir gerade betrachtet haben.

```
double[][] a;
a = new double[M][N];
for (int i = 0; i < M; i++)
   for (int j = 0; j < N; j++)
      a[i][j] = 0.0;
```

Dieser Code ist überflüssig, wenn wir mit null initialisieren, aber die verschachtelten `for`-Schleifen werden für die Initialisierung mit anderem/n Wert(en) benötigt.

Tabelle 1.4 Typischer Code für die Verarbeitung von Arrays

Aufgabe	Implementierung (Codefragment)

Ermittelt das Maximum der Arraywerte.	`double max = a[0];`
	`for (int i = 1; i < a.length; i++)`
	`  if (a[i] > max) max = a[i];`

Berechnet den Mittelwert der Arraywerte.	`int N = a.length;`
	`double sum = 0.0;`
	`for (int i = 0; i < N; i++)`
	`  sum += a[i];`
	`double average = sum / N;`

Kopiert alles in ein anderes Array.	`int N = a.length;`
	`double[] b = new double[N];`
	`for (int i = 0; i < N; i++)`
	`  b[i] = a[i];`

Kehrt die Reihenfolge der Elemente in einem Array um.	`int N = a.length;`
	`for (int i = 0; i < N/2; i++)`
	`{`
	`  double temp = a[i];`
	`  a[i] = a[N-1-i];`
	`  a[N-i-1] = temp;`
	`}`

Matrix-Matrix-Multiplikation (quadratische Matrizen) `a[][]*b[][] = c[][]`	`int N = a.length;`
	`double[][] c = new double[N][N];`
	`for (int i = 0; i < N; i++)`
	`  for (int j = 0; j < N; j++)`
	`  { // Berechnet das Skalarprodukt aus Zeile i`
	`  // und Spalte j.`
	`  for (int k = 0; k < N; k++)`
	`  c[i][j] += a[i][k]*b[k][j];`
	`  }`

1.1.6 Statische Methoden

Jedes Java-Programm in diesem Buch ist entweder eine *Datentypdefinition* (die wir ausführlich in Abschnitt 1.2 behandeln) oder eine *Bibliothek statischer Methoden* (die wir hier behandeln). Statische Methoden werden in vielen Programmiersprachen auch *Funktionen* genannt, da sie sich wie mathematische Funktionen verhalten können, was nachfolgend beschrieben werden soll. Jede statische Methode besteht aus einer Folge von

Anweisungen, die nacheinander ausgeführt werden, wenn die statische Methode aufgerufen wird. Der Modifizierer `static` unterscheidet diese Methoden von den *Instanzmethoden*, die wir in Abschnitt 1.2 besprechen. Wir verwenden den Begriff *Methode* ohne Modifizierer, wenn wir Merkmale beschreiben, die beiden Methodenarten gemeinsam sind.

Statische Methoden definieren

Eine *Methode* kapselt eine Berechnung, die als Folge von Anweisungen definiert ist. Sie übernimmt *Argumente* (Werte gegebener Datentypen) und berechnet einen *Rückgabewert*, der einem gegebenen Datentyp angehört und von den übergebenen Argumenten abhängt (beispielsweise einen Wert, der durch eine mathematische Funktion definiert ist). Oder die Methode bewirkt einen *Nebeneffekt*, der ebenfalls von den Argumenten abhängt (zum Beispiel die Ausgabe eines Wertes). Ein Beispiel für eine Methode mit einem Rückgabewert ist die statische Methode `rank()` von `BinarySearch`, ein Beispiel für eine Methode mit Nebeneffekt ist die Methode `main()`. Jede statische Methode besteht aus einer *Signatur* (die Schlüsselwörter `public static` gefolgt von einem Rückgabetyp, dem Methodennamen und einer Folge von Argumenten, jeweils mit Angabe eines deklarierten Typs) und einem *Rumpf* (einem Anweisungsblock als Folge von Anweisungen in geschweiften Klammern). Beispiele für statische Methoden finden Sie in ▶Tabelle 1.5.

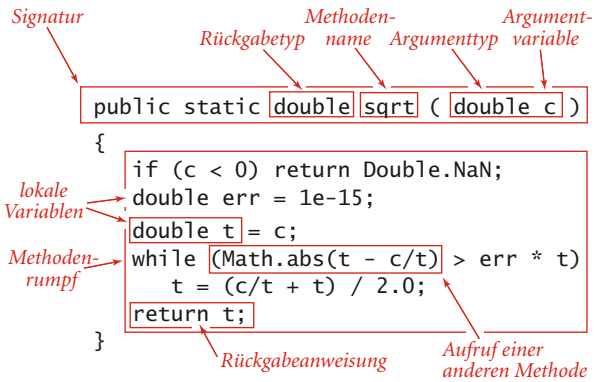


Abbildung 1.4: Aufbau einer statischen Methode

Statische Methoden aufrufen

Der *Aufruf* einer statischen Methode umfasst den Namen der Methode gefolgt von Ausdrücken in Klammern, die die Argumentwerte getrennt durch Kommata angeben. Wenn der Methodenaufruf Teil eines Ausdrucks ist, berechnet die Methode einen Wert und dieser Wert wird anstelle des Aufrufs im Ausdruck verwendet. So liefert beispielsweise der Aufruf von `rank()` von `BinarySearch` einen `int`-Wert zurück. Ein Methodenaufruf gefolgt von einem Semikolon ist eine *Anweisung*, die im Allgemeinen Nebeneffekte hat. So ist zum Beispiel der Aufruf von `Arrays.sort()` in `main()` von `BinarySearch` ein Aufruf der Systemmethode `Arrays.sort()`, die bewirkt, dass die Einträge im Array sortiert werden. Wenn eine Methode aufgerufen wird, werden die Argumentvariablen mit den Werten der

entsprechenden Ausdrücke in dem Aufruf initialisiert. Eine `return`- oder Rückgabeanweisung beendet eine statische Methode und übergibt die Kontrolle wieder an den Aufrufer. Wenn die statische Methode einen Wert berechnen soll, muss dieser Ergebniswert in einer `return`-Anweisung angegeben werden (wenn eine solche statische Methode das Ende ihrer Anweisungsfolge erreichen kann, ohne dass eine `return`-Anweisung erfolgt, gibt der Compiler eine Fehlermeldung aus).

Tabelle 1.5 Typische Implementierungen von statischen Methoden

Aufgabe	Implementierung

Absolutwert eines `int`-Wertes	`public static int abs(int x)`
	`{`
	`  if (x < 0) return -x;`
	`  else  return  x;`
	`}`

Absolutwert eines `double`-Wertes	`public static double abs(double x)`
	`{`
	`  if (x < 0.0) return -x;`
	`  else  return  x;`
	`}`

Primzahl-Test	`public static boolean isPrime(int N)`
	`{`
	`  if (N < 2) return false;`
	`  for (int i = 2; i*i <= N; i++)`
	`  if (N % i == 0) return false;`
	`  return true;`
	`}`

Quadratwurzel (Newton-Methode)	`public static double sqrt(double c)`
	`{`
	`  if (c < 0) return Double.NaN;`
	`  double err = 1e-15;`
	`  double t = c;`
	`  while (Math.abs(t - c/t) > err * t)`
	`  t = (c/t + t) / 2.0;`
	`  return t;`
	`}`

Hypotenuse eines rechtwinkligen Dreiecks	`public static double hypotenuse(double a, double b)` `{  return Math.sqrt(a*a + b*b);  }`

Harmonische Reihe (▶Tabelle 1.63 und 1.64)	`public static double H(int N)`
	`{`
	`  double sum = 0.0;`
	`  for (int i = 1; i <= N; i++)`
	`  sum += 1.0 / i;`
	`  return sum;`
	`}`

Eigenschaften von Methoden

Eine ausführliche Beschreibung der Eigenschaften von Methoden ist im Rahmen dieses Buches nicht möglich. Die folgenden Punkte sind allerdings so wichtig, dass sie nicht unerwähnt bleiben sollen:

- *Argumente werden als Wert übergeben (Pass-by-Value).* Sie können Argumentvariablen überall dort im Code des Methodenrumpfes verwenden, wo Sie auch lokale Variablen verwenden – und zwar auf die gleiche Weise. Der einzige Unterschied zwischen einer Argumentvariablen und einer lokalen Variablen ist, dass die Argumentvariable mit dem Argumentwert des aufrufenden Codes initialisiert wird. Die Methode arbeitet dabei mit dem Wert ihrer Argumente und nicht mit den Argumenten selbst. Das hat zur Folge, dass Änderungen am Wert einer Argumentvariablen in einer statischen Methode keine Auswirkungen auf den aufrufenden Code haben. In der Regel ändern wir die Argumentvariablen im Code dieses Buches nicht. Die Pass-by-Value-Konvention impliziert, dass Arrayargumente als Aliase verwendet werden (siehe Abschnitt *Aliasing* auf Seite 38) – die Methode verwendet die Argumentvariable, um auf das Array des Aufrufers Bezug zu nehmen, und kann den Inhalt des Arrays ändern (auch wenn sie das Array selbst nicht ändern kann). Zum Beispiel verändert die Methode `Arrays.sort()` den Inhalt des Arrays, das als Argument übergeben wird: Sie sortiert die Einträge.
- *Methodennamen können überladen werden.* Beispielsweise bedient sich die Java-Bibliothek `Math` dieses Ansatzes, um Implementierungen von `Math.abs()`, `Math.min()` und `Math.max()` für alle primitiven numerischen Typen anzubieten. Überladung wird aber auch häufig verwendet, um von einer Funktion zwei verschiedene Versionen zu definieren, von der eine ein Argument übernimmt und die andere einen Standardwert für dieses Argument verwendet.
- *Eine Methode hat nur einen Rückgabewert, kann aber mehrere Rückgabeanweisungen aufweisen.* Eine Java-Methode kann nur einen einzigen Rückgabewert haben, und zwar von dem Typ, der in der Methodensignatur deklariert wurde. Sobald in einer statischen Methode die erste `return`-Anweisung erreicht wird, geht die Kontrolle wieder an das aufrufende Programm zurück. Sie können `return`-Anweisungen überall dort einfügen, wo Sie sie benötigen. Doch auch wenn es mehrere `return`-Anweisungen gibt, liefert jede statische Methode bei jedem Aufruf immer nur einen Wert zurück: den Wert hinter der ersten ausgeführten `return`-Anweisung.
- *Eine Methode kann Nebeneffekte haben.* Eine Methode kann das Schlüsselwort `void` als Rückgabetyp verwenden um anzuzeigen, dass sie keinen Wert zurückliefert. In einer statischen Methode vom Typ `void` ist eine explizite `return`-Anweisung nicht erforderlich: Die Kontrolle geht automatisch nach der letzten Anweisung an den Aufrufer zurück. Statische `void`-Methoden sind bekannt dafür, dass sie Nebeneffekte produzieren (sie konsumieren Eingaben, erzeugen Ausgaben, ändern Einträge in einem Array oder ändern in irgendeiner anderen Form den Zustand des Systems). So hat zum Beispiel die statische `main()`-Methode unserer Programme `void` als Rückgabetyp, weil ihre Aufgabe darin besteht, eine Ausgabe zu erzeugen. Fachlich gesehen implementieren statische `void`-Methoden keine mathematischen Funktionen (ebenso wenig wie `Math.random()`, die keine Argumente übernimmt, aber einen Rückgabewert erzeugt).

Die Instanzmethoden, die Thema von *Kapitel 2.1* sind, weisen die gleichen Eigenschaften auf, auch wenn es hinsichtlich der Nebeneffekte fundamentale Unterschiede gibt.

Rekursion

Eine Methode kann sich selbst aufrufen (wenn Sie mit diesem als *Rekursion* bezeichneten Prinzip nicht vertraut sind, sollten Sie die *Übungen 1.1.16 bis 1.1.22* durcharbeiten). Das nachfolgende Codebeispiel zeigt eine alternative Implementierung der `rank()`-Methode aus `BinarySearch`. Wir verwenden oft rekursive Implementierungen von Methoden, da das Ergebnis kompakter, eleganter Code ist, der leichter zu verstehen ist als die entsprechende Implementierung ohne Rekursion. Der Kommentar in diesem Codebeispiel beschreibt präzise, was der Code machen soll. Wir können diesen Kommentar nutzen, um uns mittels vollständiger Induktion davon zu überzeugen, dass der Code ordnungsgemäß funktioniert. Wir werden in *Kapitel 3.1* ausführlicher auf dieses Thema eingehen und einen solchen Beweis für die binäre Suche liefern. Es gibt drei wichtige Faustregeln, die beim Entwickeln rekursiver Programme zu beachten sind:

- Die Rekursion hat eine *Abbruchbedingung* – als erste Anweisung des Programms nehmen wir immer eine bedingte Anweisung auf, die ein `return` aufweist.
- Rekursive Aufrufe müssen Teilprobleme lösen, die in gewisser Hinsicht *kleiner* sind, sodass rekursive Aufrufe gegen die Abbruchbedingung konvergieren. Im nachfolgenden Code wird die Differenz zwischen den Werten des vierten und dritten Arguments immer kleiner.
- Rekursive Aufrufe sollten keine Teilprobleme lösen, die sich *überlappen*. Im nachfolgenden Code sind die Abschnitte des Arrays, die von den beiden Teilproblemen angegangen werden, disjunkt.

Wenn eine dieser Regeln verletzt wird, müssen Sie mit fehlerhaften Ergebnissen oder einem absolut ineffizienten Programm rechnen (siehe *Übungen 1.1.19* und *1.1.27*). Beherzigen Sie jedoch die Regeln, so ist das Ergebnis ein klares und korrektes Programm, dessen Laufzeit leicht zu ermitteln ist. Ein weiterer wichtiger Grund, rekursive Methoden einzusetzen, ist, dass sie uns mathematische Modelle liefern, die uns helfen, die Laufzeit besser zu verstehen. Dieses Problem werden wir in *Kapitel 3.2* für die binäre Suche und an einigen anderen Stellen im Buch noch näher untersuchen.

```
public static int  rank(int key, int[] a)
{  return rank(key, a, 0, a.length - 1);  }
public static int rank(int key, int[] a, int lo, int hi)
{  // Index von key in a[], falls vorhanden, ist nicht kleiner als lo
   //                                       und nicht größer als hi.
   if (lo > hi) return -1;
   int mid = lo + (hi - lo) / 2;
   if      (key < a[mid]) return rank(key, a, lo, mid - 1);
   else if (key > a[mid]) return rank(key, a, mid + 1, hi);
   else                   return mid;
}
```

Listing 1.1: Rekursive Implementierung der binären Suche

Grundlegendes Programmiermodell

Eine *Bibliothek statischer Methoden* ist im Grunde ein Satz statischer Methoden, definiert in einer Java-Klasse. Dazu wird eine Datei mit den Schlüsselwörtern `public class` gefolgt von dem Klassennamen, gefolgt von den statischen Methoden in geschweiften Klammern angelegt und unter dem gleichen Namen wie die Klasse mit der Extension *.java* gespeichert. (Ein grundlegendes Java-Programmiermodell besteht darin, ein Programm, das einer bestimmten Berechnung gewidmet ist, dadurch zu entwickeln, dass man eine Bibliothek statischer Methoden aufsetzt, von denen eine `main()` heißt.) Durch die Eingabe von `java` gefolgt von einem Klassennamen, gefolgt von einer Reihe von Strings rufen Sie die `main()`-Methode aus dieser Klasse auf, und zwar mit einem Array als Argument, das die Strings enthält. Nachdem die letzte Anweisung in `main()` ausgeführt ist, wird das Programm beendet. Wenn wir in diesem Buch von einem *Java-Programm* sprechen, um eine Aufgabe zu erledigen, gehen wir von Code aus, der so oder so ähnlich entwickelt wurde (eventuell unter Hinzufügen einer Datentypdefinition, wie in Abschnitt 1.2 beschrieben). So ist `BinarySearch` beispielsweise ein Java-Programm, das aus den beiden statischen Methoden `rank()` und `main()` besteht, deren Aufgabe es ist, die Zahlen aus der Eingabe, die nicht in einer Positivliste enthalten sind, wieder auszugeben.

Modulare Programmierung

Von besonderer Bedeutung an diesem Modell ist, dass Bibliotheken statischer Methoden eine *modulare Programmierung* ermöglichen, das heißt, wir erstellen Bibliotheken von statischen Methoden (*Module*), die statische Methoden in anderen Bibliotheken aufrufen können. Dieser Ansatz hat viele Vorteile:

- Wir können mit Modulen überschaubarer Größe arbeiten, auch in Programmen, die viel Code umfassen.
- Wir können Code teilen und wiederverwenden, ohne ihn neu implementieren zu müssen.
- Wir können Implementierungen problemlos durch verbesserte Implementierungen ersetzen.
- Wir können geeignete abstrakte Modelle entwickeln, um Programmierprobleme zu bewältigen.
- Wir können das Debuggen eingrenzen (siehe den nachfolgenden Abschnitt zu den Modultests).

Zum Beispiel macht `BinarySearch` Gebrauch von drei anderen unabhängig voneinander entwickelten Bibliotheken: unsere Bibliotheken `StdIn` und `In` und die Java-Bibliothek `Arrays`. Jede dieser Bibliotheken macht wiederum Gebrauch von weiteren Bibliotheken.

Modultests

In der Java-Programmierung ist es allgemein üblich, jede Bibliothek von statischen Methoden mit einer `main()`-Methode zu versehen, die die Methoden in der Bibliothek testet (einige Programmiersprachen verbieten mehrere `main()`-Methoden und unterstützen diesen Ansatz nicht). Anspruchsvolle Modultests können eine große Programmierherausforderung sein. Jedes Modul sollte mindestens eine `main()`-Methode enthalten, die den Code in dem Modul ausführt und gewährleistet, dass er funktioniert. Im Laufe der Erweiterung eines Moduls nehmen wir oft Verbesserungen an der `main()`-Methode vor und bauen sie entweder zu einem *Entwicklungsclient* aus, der uns hilft, den Code während der Entwicklung im Detail zu testen, oder zu einem *Testclient*, der den gesamten Code ausgiebig testet. Wird der Client komplizierter, können wir ihn in einem eigenen unabhängigen Modul kapseln. In diesem Buch verwenden wir `main()`, um Ihnen den Zweck jedes Moduls zu veranschaulichen, und überlassen Ihnen die Testclients als Übung.

Externe Bibliotheken

Wir verwenden statische Methoden aus vier verschiedenen Arten von Bibliotheken, die jeweils (leicht) abweichende Prozeduren für die Wiederverwendung von Code erfordern. Die meisten dieser Bibliotheken bestehen ausschließlich aus statischen Methoden, einige sind jedoch Datentypdefinitionen, die auch einige statische Methoden umfassen.

- Die Standardsystembibliotheken `java.lang.*`: Hierzu gehören `Math`, die Methoden für häufig verwendete mathematische Funktionen enthält, `Integer` und `Double`, die wir nutzen, um Zeichenketten in `int`- und `double`-Werte oder vice versa umzuwandeln, `String` und `StringBuilder`, die wir weiter hinten in diesem Abschnitt und in *Kapitel 5* noch näher besprechen, sowie Dutzende von weiteren Bibliotheken, die wir hier nicht verwenden.
- Importierte Systembibliotheken wie `java.util.Arrays`: Es gibt Tausende solcher Bibliotheken in einem Standard-Java-Release, die wir allerdings nur sehr begrenzt nutzen. Sie können auf solche Bibliotheken erst zugreifen, nachdem Sie sie mit einer `import`-Anweisung am Anfang des Programms eingebunden haben.
- Andere Bibliotheken in diesem Buch: Zum Beispiel kann ein anderes Programm die Methode `rank()` von `BinarySearch` nutzen. Um solch ein Programm zu verwenden, laden Sie den Quellcode einfach von der Website zum Buch in Ihr Arbeitsverzeichnis herunter.
- Die Standardbibliotheken `Std*`, die wir speziell für dieses Buch (und unser Einsteigerbuch *Einführung in die Programmierung mit Java: Ein interdisziplinärer Ansatz*) entwickelt haben: Diese Bibliotheken werden in den folgenden Seiten vorgestellt. Den Quellcode und Anweisungen zum Herunterladen finden Sie auf der Website zum Buch.

Tabelle 1.6 Die in diesem Buch verwendeten Bibliotheken mit statischen Methoden

Standardsystembibliotheken

`Math`
`Integer`+
`Double`+
`String`+
`StringBuilder`
`System`

Importierte Systembibliotheken

`java.util.Arrays`

Unsere Standardbibliotheken

`StdIn`
`StdOut`
`StdDraw`
`StdRandom`
`StdStats`
`In`+
`Out`+

+ Datentypdefinitionen, die einige statische Methoden enthalten.

Um eine Methode einer anderen Bibliothek aufzurufen (sei dies eine Bibliothek im gleichen oder einem explizit angegebenen Verzeichnis, eine Standardsystembibliothek oder eine Systembibliothek, die vor der Klassendefinition mittels einer `import`-Anweisung eingebunden wird), setzen wir bei jedem Aufruf den Bibliotheksnamen vor den Namen der Methode. So ruft zum Beispiel die `main()`-Methode von `BinarySearch` die `sort()`-Methode der Systembibliothek `java.util.Arrays`, die `readInts()`-Methode unserer Bibliothek `In` und die `println()`-Methode unserer Bibliothek `StdOut` auf.

Bibliotheken von Methoden, die wir selbst oder andere in einer modularen Programmierumgebung implementiert haben, können den Anwendungsbereich unseres Programmiermodells extrem erweitern. So gibt es zusätzlich zu den Bibliotheken des Standard-Java-Release noch Tausende von weiteren Bibliotheken im Web zu Anwendungen aller Arten. Da wir uns in diesem Buch aber auf Algorithmen konzentrieren wollen, sind wir bemüht, den Anwendungsbereich unseres Programmiermodells überschaubar zu halten, und verwenden deshalb nur die Bibliotheken aus ▶Tabelle 1.6, und davon auch nur eine Teilmenge der Methoden, die in den nachfolgend beschriebenen APIs aufgelistet sind.

1.1.7 APIs

Eine wichtige Komponente der modularen Programmierung ist die *Dokumentation*. Sie erläutert die Funktionsweise der Bibliotheksmethoden, die anderen zur Nutzung zur Verfügung stehen. Wir werden konsequent alle Bibliotheksmethoden, die wir in diesem Buch verwenden, in sogenannten *Programmierschnittstellen* oder *APIs* (für *Applications Programming Interface*) beschreiben – unter Angabe des Bibliotheksnamens, der Signatur und einer Kurzbeschreibung. Mit dem Begriff *Client* beziehen wir uns auf ein Programm, das eine Methode einer anderen Bibliothek aufruft, und der Begriff *Implementierung* beschreibt den Java-Code, der die Methoden in einer API implementiert.

Beispiel

Das folgende Beispiel – die API für häufig verwendete statische Methoden der Standardbibliothek `Math` in `java.lang` – soll unsere Konventionen für APIs veranschaulichen:

Tabelle 1.7 API für die Java-Bibliothek `Math` (Auszug)

public class Math

`static double abs(double a)`	Absolutwert von `a`
`static double max(double a, double b)`	Maximum von `a` und `b`
`static double min(double a, double b)`	Minimum von `a` und `b`
Hinweis 1: `abs()`, `max()` und `min()` sind auch für `int`, `long` und `float` definiert.	
`static double sin(double theta)`	Sinusfunktion
`static double cos(double theta)`	Kosinusfunktion
`static double tan(double theta)`	Tangensfunktion
Hinweis 2: Winkel werden in Bogenmaß angegeben. Verwenden Sie für die Umwandlung `toDegrees()` und `toRadians()`.	
Hinweis 3: Verwenden Sie `asin()`, `acos()` und `atan()` für die Umkehrfunktionen.	
`static double exp(double a)`	Exponentialfunktion (e^a)
`static double log(double a)`	Natürlicher Logarithmus ($\log_e a$ oder ln `a`)
`static double pow(double a, double b)`	`a` hoch `b` (a^b)
`static double random()`	Zufallszahl zwischen [0, 1)
`static double sqrt(double a)`	Quadratwurzel von `a`
`static double E`	Wert von `e` (Konstante)
`static double PI`	Wert von π (Konstante)
Auf der Website zum Buch finden Sie weitere Funktionen, auf die Sie zurückgreifen können.	

Alle diese Methoden implementieren mathematische Funktionen, d.h., sie nehmen Argumente entgegen und berechnen daraus einen Ergebniswert eines vorgegebenen Typs (außer `random()`, die keine mathematische Funktion implementiert, weil sie keine Argumente übernimmt). Da alle Methoden `double`-Werte einsetzen und ein `double`-Ergebnis berechnen, können Sie sie sozusagen als Erweiterung des Datentyps `double` betrachten – eine solche Erweiterbarkeit ist eines der Hauptmerkmale moderner Programmiersprachen. Jede Methode wird in der API mit nur einer Zeile beschrieben, die alle Informationen enthält, die Sie benötigen, um diese Methode einsetzen zu können. Die Bibliothek `Math` definiert außerdem die relativ genauen konstanten Werte `PI` (für π) und `E` (für e), sodass Sie diese Namen in Ihren Programmen verwenden können, um auf diese Konstanten Bezug zu nehmen. So ist zum Beispiel der Wert von `Math.sin(Math.PI/2)` gleich `1.0` und der Wert von `Math.log(Math.E)` gleich `1.0` (da `Math.sin()` sein Argument als Bogenmaß übernimmt und `Math.log()` die natürliche Logarithmusfunktion implementiert).

Java-Bibliotheken

Jedes Java-Release umfasst ausführliche Onlinebeschreibungen zu Tausenden von Bibliotheken. Wir beschränken uns in diesem Buch aber auf nur einige wenige Methoden, um unser Programmiermodell möglichst übersichtlich zu halten. Zum Beispiel verwendet `BinarySearch` die Methode `sort()` aus der Java-Bibliothek `Arrays`, die wir folgendermaßen dokumentieren:

Tabelle 1.8 Auszug aus der Java-Bibliothek `Arrays` (`java.util.Arrays`)

public class Arrays	
`static void sort(int[] a)`	Das Array in aufsteigender Reihenfolge sortieren
Hinweis: Diese Methode ist auch für andere primitive Datentypen und `Object` definiert.	

Da die Bibliothek `Arrays` nicht Teil von `java.lang` ist, wird eine `import`-Anweisung benötigt, um eine Verwendung wie in `BinarySearch` zu ermöglichen. In *Kapitel 2* dieses Buches gehen wir ausführlich auf die Implementierungen von `sort()` für Arrays ein, einschließlich der Mergesort- und Quicksort-Algorithmen, die in `Arrays.sort()` implementiert sind. Viele der grundlegenden Algorithmen, die wir in diesem Buch betrachten, sind in Java und vielen anderen Programmierumgebungen implementiert – wie zum Beispiel `Arrays`, das eine Implementierung der binären Suche enthält. Um Verwirrungen zu vermeiden, verwenden wir im Allgemeinen unsere eigenen Implementierungen, obwohl nichts dagegen spricht, auf gut abgestimmte Bibliotheksimplementierungen zurückzugreifen, sofern man diese richtig verstanden hat.

Unsere Standardbibliotheken

Wir haben eine Reihe von Bibliotheken entwickelt, deren Funktionalität vor allem für den Einstieg in die Java-Programmierung, für wissenschaftliche Anwendungen sowie für das Entwickeln, Studieren und Anwenden von Algorithmen von Nutzen ist. Die

meisten dieser Bibliotheken sind mit der Ein- und Ausgabe verbunden. Darüber hinaus arbeiten wir mit den folgenden beiden Bibliotheken, um unsere Implementierungen zu testen und zu analysieren. Die erste erweitert `Math.random()`, sodass wir Zufallswerte von verschiedenen Verteilungen erzeugen können, die zweite unterstützt statistische Berechnungen:

Tabelle 1.9 API für unsere Bibliothek der statischen Methoden für Zufallszahlen

Public class StdRandom

`static`	`void`	`setSeed(long seed)`	Initialisieren
`static`	`double`	`random()`	Reelle Zahl zwischen `0` und `1`
`static`	`int`	`uniform(int N)`	Integer zwischen `0` und `N-1`
`static`	`int`	`uniform(int lo, int hi)`	Integer zwischen `lo` und `hi-1`
`static`	`double`	`uniform(double lo, double hi)`	Reelle Zahl zwischen `lo` und `hi`
`static`	`boolean`	`bernoulli(double p)`	`true` mit der Wahrscheinlichkeit `p`
`static`	`double`	`gaussian()`	Normal, Mittelwert `0`, Standardabweichung `1`
`static`	`double`	`gaussian(double m, double s)`	Normal, Mittelwert `m`, Standardabweichung `s`
`static`	`int`	`discrete(double[] a)`	`i` mit der Wahrscheinlichkeit `a[i]`
`static`	`void`	`shuffle(double[] a)`	Mischen des Arrays `[a]`

Hinweis: Es gibt überladene Implementierungen von `shuffle()` für andere primitive Datentypen und `Object`.

Tabelle 1.10 API für unsere Bibliothek der statischen Methoden zur Datenanalyse

public class StdStats

`static`	`double`	`max(double[] a)`	Größter Wert
`static`	`double`	`min(double[] a)`	Kleinster Wert
`static`	`double`	`mean(double[] a)`	Mittelwert
`static`	`double`	`var(double[] a)`	Stichprobenvarianz
`static`	`double`	`stddev(double[] a)`	Stichproben-Standardabweichung
`static`	`double`	`median(double[] a)`	Zentralwert

Die Methode `setSeed()` in `StdRandom` erlaubt die Initialisierung des Zufallszahlengenerators, damit wir Experimente mit Zufallszahlen wiederholen können. (In ▶Tabelle 1.11 sind für viele dieser Methoden die Implementierungen als Referenz angegeben.) Einige dieser Methoden sind extrem leicht zu implementieren. Warum also sich die Mühe machen, sie in eine Bibliothek aufzunehmen? Für gut entworfene Bibliotheken lauten die Antworten auf diese Frage normalerweise:

- Die Methoden implementieren eine Abstraktionsebene, die es uns erlaubt, uns auf das Implementieren und Testen der Algorithmen im Buch zu konzentrieren, und nicht auf das Erzeugen von Zufallsobjekten oder das Berechnen statistischer Werte. Client-Code, der diese Methoden verwendet, ist klarer und verständlicher als selbst aufgesetzter Code, der die gleichen Berechnungen anstellt.
- Bibliotheksimplementierungen testen auf außergewöhnliche Bedingungen, decken seltene Situationen ab und sind intensiv getestet, sodass wir uns auf ein einwandfreies Funktionieren verlassen können. Solche Implementierungen bestehen teilweise aus sehr viel Code – auch weil man manchmal für verschiedene Datentypen gerne individuelle Implementierungen hätte. Aus diesem Grund enthält die Java-Bibliothek `Arrays` mehrere überladene Implementierungen von `sort()` – eine für jeden Datentyp, den Sie möglicherweise sortieren müssen.

Solche Überlegungen sind von grundlegender Bedeutung für die modulare Programmierung in Java, aber in diesem Fall vielleicht etwas übertrieben. Auch wenn die Methoden der beiden Bibliotheken mehr oder weniger selbsterklärend sind und viele davon sich relativ leicht implementieren lassen, sind einige von ihnen interessante algorithmische Übungen. Deshalb täten Sie gut daran, den Code in `StdRandom.java` und `StdStats.java` auf der Website zum Buch *nicht nur* zu studieren, *sondern* diese bewährten Implementierungen auch direkt einzusetzen. Wer diese Bibliotheken nutzen (und den Code untersuchen) möchte, sollte am besten den Quellcode von der Website zum Buch herunterladen und ihn in seinem Arbeitsverzeichnis ablegen. Auf der Website finden Sie außerdem die Beschreibung mehrerer systemabhängiger Mechanismen, wie Sie diese Bibliotheken nutzen können, ohne unnötig Kopien zu erstellen.

Ihre eigenen Bibliotheken

Es empfiehlt sich, *jedes Programm, das Sie schreiben*, als Bibliotheksimplementierung zu betrachten, die wiederverwendet werden kann.

- Schreiben Sie Code für den Client – eine Toplevel-Implementierung, die die Berechnung in leichter zu handhabende Teile aufbricht.
- Formulieren Sie eine API für eine Bibliothek von statischen Methoden (oder mehrere APIs für mehrere Bibliotheken), mit denen sich die einzelnen Teile lösen lassen.
- Entwickeln Sie eine Implementierung der API, mit einer `main()`-Methode, die die Methoden unabhängig vom Client testet.

Mit diesem Ansatz entwickeln Sie nicht nur wertvolle Software, die Sie später wiederverwenden können, sondern machen auch bestmöglichen Gebrauch von der modularen Programmierung – der Schlüssel zur erfolgreichen Lösung komplexer Programmieraufgaben.

Tabelle 1.11 Implementierungen von statischen Methoden der Bibliothek `StdRandom`

Angestrebtes Ergebnis	Implementierung
`double`-Zufallswert in `[a, b)`	`public static double uniform(double a, double b)` `{ return a + StdRandom.random() * (b-a); }`
`int`-Zufallswert in `[0..N)`	`public static int uniform(int N)` `{ return (int) (StdRandom.random() * N); }`
`int`-Zufallswert in `[lo..hi)`	`public static int uniform(int lo, int hi)` `{ return lo + StdRandom.uniform(hi - lo); }`

`int`-Zufallswert aus einer diskreten Verteilung (`i` mit der Wahrscheinlichkeit `a[i]`)	`public static int discrete(double[] a)`
	`{ // Elemente in a[] müssen als Summe 1 ergeben.`
	`double r = StdRandom.random();`
	`double sum = 0.0;`
	`for (int i = 0; i < a.length; i++)`
	`{`
	`sum = sum + a[i];`
	`if (sum >= r) return i;`
	`}`
	`return -1;`
	`}`

Mischen der Elemente in einem Array von `double`-Werten (siehe Übung 1.1.36)	`public static void shuffle(double[] a)`
	`{`
	`int N = a.length;`
	`for (int i = 0; i < N; i++)`
	`{ // Tausche a[i] mit Zufallselement in a[i..N-1]`
	`int r = i + StdRandom.uniform(N-i);`
	`double temp = a[i];`
	`a[i] = a[r];`
	`a[r] = temp;`
	`}`
	`}`

Sinn und Zweck einer API ist es, den Client von der Implementierung zu *trennen*: Der Client sollte von der Implementierung lediglich das wissen, was die API an Informationen bietet, und die Implementierung sollte keine Eigenschaften von irgendwelchen Clients berücksichtigen. APIs bieten uns die Möglichkeit, Code für die verschiedensten Anwendungsbereiche zu entwickeln und dann vielfach wiederzuverwenden. Keine Java-Bibliothek kann alle Methoden enthalten, die wir zur Lösung eines komplexen Problems benötigen, sodass die Wiederverwendung ein wichtiger Schritt zur Lösung komplexer Aufgaben ist. Programmierer betrachten eine API als eine Art *Vertrag* zwischen dem Client und der Implementierung, der klar angibt, was jede Methode macht. Wenn wir an einer Implementierung arbeiten, ist es unser oberstes Ziel, die Vertragsbedingungen einzuhalten. Oft kann dies auf verschiedenen Wegen erreicht werden, und wir haben dank der Trennung von Client-Code und Implementierung die Freiheit, im Laufe

der Zeit immer wieder ältere Implementierungen durch neue, verbesserte Versionen zu ersetzen. Beim Studium der Algorithmen ist diese Möglichkeit von großer Bedeutung, um festzustellen, inwieweit sich unsere Verbesserungen an den Algorithmen auswirken.

1.1.8 Strings

Ein `String` ist eine Folge von Zeichen (`char`-Werten). Ein `String`-Literal ist eine Folge von Zeichen in doppelten Anführungszeichen, wie `"Hello, World"`. Der Datentyp `String` ist zwar ein integrierter, aber *kein* primitiver Datentyp, auch wenn Java ihn manchmal so behandelt. Wir gehen schon hier auf `String` ein, da es ein wichtiger Datentyp ist, den fast jedes Java-Programm verwendet.

Verkettung

In Java gibt es neben den integrierten Operatoren für primitive Datentypen auch einen integrierten Verkettungsoperator (+) für `String`, weswegen man die Zeile aus ▶Tabelle 1.12 auch in Tabelle 1.2 der primitiven Datentypen aufnehmen könnte. Das Ergebnis beim Verketten zweier `String`-Werte ist ein einzelner `String`-Wert: die erste Zeichenkette gefolgt von der zweiten.

Tabelle 1.12 Der Java-Datentyp `String`

Typ	Wertesatz	Typische Literale	Operatoren	Typische Ausdrücke	
				Ausdruck	Wert

`String`	Zeichenfolgen	`"AB"`
		`"Hello"`
		`"2.5"`
			`+` (verketten)	`"Hi, " + "Bob"`
				`"12" + "34"`
				`"1" + "+" + "2"`
					`"Hi, Bob"`
					`"1234"`
					`"1+2"`

Umwandlung

Strings spielen vor allem in zwei Situationen eine wichtige Rolle: um Werte, die wir über eine Tastatur eintippen, in Datentypwerte umzuwandeln und um Datentypwerte in Werte umzuwandeln, die wir auf einer Anzeige lesen können. Java verfügt in seinen Bibliotheken bereits über Methoden für `String`, die diese Operationen erleichtern. Erwähnenswert sind hierbei die Bibliotheken `Integer` und `Double` mit ihren statischen Methoden, um `String`-Werte in `int`-Werte (und umgekehrt) bzw. `String`-Werte in `double`-Werte (und umgekehrt) umzuwandeln.

Tabelle 1.13 APIs für die Umwandlung zwischen Zahlen und `String`-Werten

public class Integer			
`static`	`int`	`parseInt(String s)`	Wandelt `s` in einen `int`-Wert um.
`static`	`String`	`toString(int i)`	Wandelt `i` in einen `String`-Wert um.

Tabelle 1.13 APIs für die Umwandlung zwischen Zahlen und `String`-Werten *(Forts.)*

public class Double

`static`	`double`	`parseDouble(String s)`	Wandelt `s` in einen `double`-Wert um.
`static`	`String`	`toString(double x)`	Wandelt `x` in einen `String`-Wert um.

Automatische Umwandlung

Wir rufen die oben beschriebenen statischen `toString()`-Methoden nur selten auf, da Java über einen eingebauten Mechanismus verfügt, um durch Verkettung einen beliebigen Datentypwert in einen `String`-Wert umzuwandeln: Wenn *einer* der Argumente von `+` ein `String` ist, wandelt Java *automatisch* das andere Argument ebenfalls in einen `String` um (sofern es nicht bereits ein `String` ist). Dieser Mechanismus erlaubt uns nicht nur Formulierungen wie `"Die Quadratwurzeln von 2.0 ist " + Math.sqrt(2.0)`, sondern ermöglicht es uns auch, einen Wert jeden beliebigen Datentyps durch Verkettung mit dem leeren String `""` in einen `String` umzuwandeln.

Befehlszeilenargumente

Strings kommen in der Java-Programmierung noch eine besondere Rolle zu, weil sie an dem Mechanismus beteiligt sind, mit dem Daten von der Befehlszeile an das Programm übergeben werden. Der Mechanismus selbst ist relativ simpel. Wenn Sie den Befehl `java` mit einem Bibliotheksnamen gefolgt von einem oder mehreren Strings eintippen, ruft das Java-System die Methode `main()` in dieser Bibliothek auf und übergibt ihr ein *Array von Strings* als Argument: eben die Strings, die auf den Bibliotheksnamen folgen. Die `main()`-Methode von `BinarySearch` übernimmt z.B. genau ein Befehlszeilenargument, sodass das System für die Übergabe ein Array der Größe eins erzeugt. Das Programm verwendet diesen Wert, `args[0]`, um die Datei mit der Positivliste zu benennen, die als Argument an `In.readInts()` übergeben wird. Für den Fall, dass ein Befehlszeilenargument eine Zahl repräsentieren soll, haben wir es mit einem weiteren typischen Paradigma zu tun, das wir oft in unserem Code verwenden: mit `parseInt()` wandeln wir das Argument in einen `int`-Wert um und mit `parseDouble()` in einen `double`-Wert.

Rechnen mit Strings ist ein wichtiger Bestandteil moderner Programmierung. Im Moment nutzen wir `String` nur, um zwischen der externen Repräsentation von Zahlen als Zeichenfolgen und der internen Repräsentation numerischer Datentypwerte umzuwandeln. In Abschnitt 1.2 werden wir sehen, dass Java viele weitere Operationen auf `String`-Werten unterstützt, die wir in diesem Buch ebenfalls verwenden. In Abschnitt 1.4 werden wir uns mit der internen Repräsentation von `String`-Werten beschäftigen und in *Kapitel 5* werden wir ausführlich auf Algorithmen eingehen, die `String`-Daten verarbeiten. Diese Algorithmen sind die interessantesten, kompliziertesten und bedeutsamsten Methoden, die wir in diesem Buch betrachten.

1.1.9 Ein- und Ausgabe

Unsere Standardbibliotheken für Eingabe, Ausgabe und Grafik sind vornehmlich dazu gedacht, unseren Java-Programmen ein einfaches Modell für die Kommunikation mit der Außenwelt zur Verfügung zu stellen. Diese Bibliotheken stützen sich dabei auf die umfangreichen Möglichkeiten, die die Java-Bibliotheken bieten, dort aber im Allgemeinen relativ kompliziert und schwer zu verstehen und zu nutzen sind. Sehen wir uns zunächst kurz an, wie das Modell aufgebaut ist.

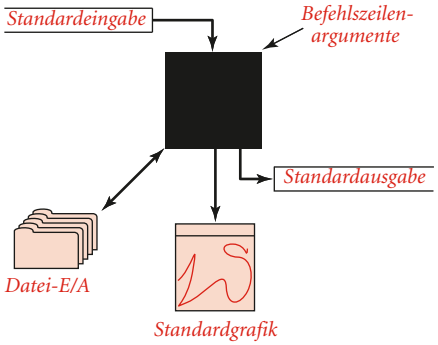


Abbildung 1.5: Ein Java-Programm aus der Vogelperspektive

In unserem Modell nimmt ein Java-Programm Eingabewerte aus *Befehlszeilenargumenten* oder einem abstrakten Zeichenstrom (der sogenannten *Standardeingabe*) entgegen und schreibt in einen anderen abstrakten Zeichenstrom, der sogenannten *Standardausgabe*.

Aus diesem Grund müssen wir die Schnittstelle zwischen Java und dem Betriebssystem näher untersuchen, das heißt, wir müssen auf die grundlegenden Mechanismen der meisten modernen Betriebssysteme und Entwicklungsumgebungen eingehen. Wenn Sie ausführlichere Informationen zu einem speziellen Betriebssystem benötigen, finden Sie diese auf der Website zum Buch. Standardmäßig sind Befehlszeilenargumente sowie Standardeingabe und -ausgabe mit Anwendungen verbunden, die entweder durch das Betriebssystem unterstützt werden oder durch eine Programmentwicklungsumgebung, die die Befehle entgegennimmt. Das Fenster der Anwendung, in das wir Text eingeben und in dem wir Textausgaben lesen, bezeichnen wir allgemein als *Konsolenfenster*. Seit den ersten Unix-Systemen in den 1970ern hat sich dieses Modell bewährt, um auf einem bequemen und direkten Weg mit unseren Programmen und Daten zu interagieren. Wir ergänzen das klassische Modell um die *Standardgrafik*, sodass wir unsere Daten zur Datenanalyse auch visuell aufbereiten können.

Befehle und Argumente

Das Konsolenfenster zeigt eine Eingabeaufforderung (*prompt)*, hinter dem wir *Befehle* an das Betriebssystem eingeben, je nach Bedarf mit oder ohne *Argumente*. Wir verwenden nur sehr wenige Befehle in diesem Buch, die kurz in ▶Tabelle 1.14 zusammengefasst werden. Den Befehl, den wir mit Sicherheit am häufigsten verwenden, ist der Befehl `java`, mit dem wir unsere Programme ausführen. Wie in Abschnitt *Befehlszeilenargumente* erwähnt, haben Java-Klassen eine statische Methode `main()`, die ein `String`-Array `args[]` als Argument übernimmt. Dieses Array enthält die Folge der von uns eingegebenen Befehlszeilenargumente, die das Betriebssystem an Java weiterleitet. Per Konvention verarbeiten Java und das Betriebssystem die Argumente als Strings. Wenn eines der Argumente als Zahl interpretiert werden soll, rufen wir eine Methode wie `Integer.parseInt()` auf, um es von einem `String` in den entsprechenden Datentyp umzuwandeln.

Tabelle 1.14 Typische Betriebssystembefehle

Befehl	Argumente	Zweck
`javac`	Name einer `.java`-Datei	Kompiliert ein Java-Programm.
`java`	Name einer `.class`-Datei (ohne Erweiterung) und Befehlszeilenargumente	Führt ein Java-Programm aus.
`more`	Jeder Textdateiname	Gibt den Dateiinhalt aus.

Standardausgabe

Unsere Bibliothek `StdOut` unterstützt die Standardausgabe. Per Voreinstellung verbindet das Betriebssystem die Standardausgabe mit dem Konsolenfenster. Die Methode `print()` gibt die ihr übergebenen Argumente auf der Standardausgabe aus. Die Methode `println()` fügt einen Zeilenumbruch hinzu und die Methode `printf()` unterstützt die nachfolgend beschriebene formatierte Ausgabe. Java verfügt über eine ähnliche Methode in ihrer Bibliothek `System.out`. Wir verwenden jedoch `StdOut`, um Standardeingabe und Standardausgabe gleich zu behandeln (und um einige technische Verbesserungen vorzunehmen).

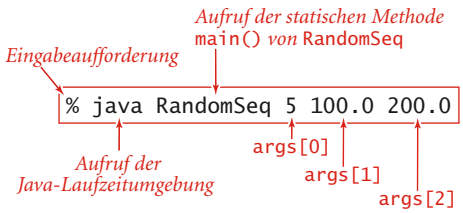


Abbildung 1.6: Aufbau eines Befehls

Tabelle 1.15 API für unsere Bibliothek der statischen Standardausgabemethoden

public class StdOut	
`static void print(String s)`	Gibt `s` aus.
`static void println(String s)`	Gibt `s` gefolgt von einem Zeilenumbruch aus.
`static void println()`	Gibt einen Zeilenumbruch aus.
`static void printf(String f, ...)`	Formatiert die Ausgabe.
Hinweis: Für primitive Datentypen und für `Object` gibt es überladene Implementierungen.	

Um diese Methoden zu nutzen, müssen Sie *StdOut.java* von der Website zum Buch in Ihr Arbeitsverzeichnis herunterladen und die Methoden dann mit Code wie `StdOut.println("Hello, World");` aufrufen. In ▶Listing 1.2 wird ein Beispielclient vorgestellt.

```
public class RandomSeq
{
   public static void main(String[] args)
   {  // Gibt N Zufallswerte aus dem Bereich (lo, hi) aus.
      int N = Integer.parseInt(args[0]);
      double lo = Double.parseDouble(args[1]);
      double hi = Double.parseDouble(args[2]);
      for (int i = 0; i < N; i++)
      {
         double x = StdRandom.uniform(lo, hi);
         StdOut.printf("%.2f\n", x);
      }
   }
}
```

Listing 1.2: `StdOut`-Beispielclient

```
% java RandomSeq 5 100.0 200.0
123.43
153.13
144.38
155.18
104.02
```

Formatierte Ausgabe

In ihrer einfachsten Form übernimmt die Methode `printf()` zwei Argumente. Das erste Argument ist ein *Formatstring*, der beschreibt, wie das zweite Argument für die Ausgabe in einen String umzuwandeln ist. Der einfachste Formatstring beginnt mit einem `%` und endet mit einem einbuchstabigen *Umwandlungscode*. Die geläufigsten Umwandlungscodes sind `d` (für Dezimalwerte aus Javas Integertypen), `f` (für Gleitkommawerte) und `s` (für `String`-Werte). Zwischen dem `%` und dem Umwandlungscode steht

ein Integer, der die *Feldbreite* des umgewandelten Wertes angibt (die Anzahl der Zeichen in dem erzeugten Ausgabestring). Standardmäßig werden linksseitig Leerzeichen eingefügt, um die Länge der erzeugten Ausgabe an die gewünschte Feldbreite anzupassen. Wenn die Leerzeichen rechts stehen sollen, können wir ein Minuszeichen vor die Feldbreite setzen. (Wenn der umgewandelte Ausgabestring größer als die Feldbreite ist, wird die Feldbreite ignoriert.) Im Anschluss an die Breite haben wir die Möglichkeit, einen Punkt und dann – für `double`-Werte – die Anzahl der Ziffern nach dem Dezimalpunkt (die Genauigkeit) beziehungsweise – für `String`-Werte – die Anzahl der Zeichen ab Beginn des Strings einzugeben. Das Wichtigste, was Sie sich für `printf()` merken müssen, ist, dass *der Umwandlungscode im Formatstring und der Typ des dazugehörigen Arguments übereinstimmen müssen*. Das bedeutet, Java muss in der Lage sein, vom Typ des Arguments in den vom Umwandlungscode geforderten Typ umzuwandeln. Das erste Argument von `printf()` ist ein `String`, der neben einem Formatstring auch andere Zeichen enthalten kann. Jeder Teil des Arguments, der nicht Teil eines Formatstrings ist, wird direkt an die Ausgabe weitergereicht, während der Formatstring durch den Argumentwert ersetzt wird (wie angegeben, umgewandelt in einen String). So gibt zum Beispiel die Anweisung

```
StdOut.printf("PI ist ungefähr %.2f\n", Math.PI);
```

die folgende Zeile aus:

```
PI ist ungefähr 3.14
```

Beachten Sie, dass wir im Argument explizit ein Zeilenumbruch-Zeichen (`\n`) angeben müssen, um mit `printf()` einen Zeilenumbruch auszugeben. Die `printf()`-Funktion kann mehr als zwei Argumente übernehmen. In diesem Fall enthält der Formatstring einen Formatspezifizierer für jedes weitere Argument, optional getrennt durch weitere, direkt an die Ausgabe weiterzuleitende Zeichen. Wenn Sie der statischen Methode `String.format()` die gleichen Argumente übergeben wie gerade `printf()`, erhalten Sie einen formatierten String, ohne ihn auszugeben. Die formatierte Ausgabe ist ein bequemer Mechanismus, um mithilfe von kompaktem Code Messdaten tabellarisch aufzubereiten (unser Hauptzweck in diesem Buch).

Tabelle 1.16 Formatkonventionen für `printf()` (viele weitere Optionen finden Sie auf der Website zum Buch)

Typ	Code	Typisches Literal	Formatstring-Beispiele	Konvertierte String-Werte für die Ausgabe
`int`	`d`	`512`	`"%14d"` `"%-14d"`	`"  512"` `"512  "`
`double`	`f`	`1595.1680010754388`	`"%14.2f"` `"%.7f"`	`"  1595.17"` `"1595.1680011"`
	`e`		`"%14.4e"`	`"  1.5952e+03"`

`String`	`s`	`"Hello, World"`	`"%14s"`
			`"%-14s"`
			`"%-14.5s"`
				`"  Hello, World"`
				`"Hello, World  "`
				`"Hello  "`

Standardeingabe

Unsere `StdIn`-Bibliothek übernimmt die Daten aus der *Standardeingabe*, der leer sein oder eine Folge von Werten enthalten kann, die durch Whitespace (Leerzeichen, Tabulatoren, Zeilenumbruchzeichen usw.) getrennt sind. Per Voreinstellung verbindet das Betriebssystem die Standardausgabe mit dem Konsolenfenster – was Sie eingeben, ist der Eingabestrom (der je nach Ihrer Konsolenanwendung mit (Strg)+(d) oder (Strg)+(z) beendet wird). Jeder Wert ist ein `String` oder ein Wert eines primitiven Java-Typs. Eines der Hauptmerkmale der Standardeingabe ist, dass Ihr Programm die Werte beim Lesen *verbraucht*. Sobald Ihr Programm einen Wert gelesen hat, kann es nicht mehr zurück und den Wert erneut lesen. Dieses Verhalten ist recht restriktiv, spiegelt aber die physikalischen Eigenschaften einiger Eingabegeräte wider und vereinfacht das Implementieren der Abstraktion. Im Rahmen des Eingabestrommodells sind die statischen Methoden dieser Bibliothek selbsterklärend (beschrieben durch ihre Signaturen).

```
public class Average
{
   public static void main(String[] args)
   {  // Berechnet den Mittelwert der Zahlen in StdIn.
      double sum = 0.0;
      int cnt = 0;
      while (!StdIn.isEmpty())
      {  // Liest eine Zahl und addiert sie zu der Summe.
         sum += StdIn.readDouble();
         cnt++;
      }
      double avg = sum / cnt;
      StdOut.printf("Average is %.5f\n", avg);
   }
}
```

Listing 1.3: `StdIn`-Beispielclient

```
% java Average
1.23456
2.34567
3.45678
4.56789
<ctrl-d>
Average is 2.90123
```

Tabelle 1.17 API für unsere Bibliothek der statischen Standardeingabemethoden

public class StdIn			
`static`	`boolean`	`isEmpty()`	`true`, wenn es keine weiteren Werte gibt, ansonsten `false`.
`static`	`int`	`readInt()`	Liest einen Wert vom Typ `int`.

Tabelle 1.17 API für unsere Bibliothek der statischen Standardeingabemethoden *(Forts.)*

public class StdIn			
`static`	`double`	`readDouble()`	Liest einen Wert vom Typ `double`.
`static`	`float`	`readFloat()`	Liest einen Wert vom Typ `float`.
`static`	`long`	`readLong()`	Liest einen Wert vom Typ `long`.
`static`	`boolean`	`readBoolean()`	Liest einen Wert vom Typ `boolean`.
`static`	`char`	`readChar()`	Liest einen Wert vom Typ `char`.
`static`	`byte`	`readByte()`	Liest einen Wert vom Typ `byte`.
`static`	`String`	`readString()`	Liest einen Wert vom Typ `String`.
`static`	`boolean`	`hasNextLine()`	Gibt es eine weitere Zeile im Eingabestrom?
`static`	`String`	`readLine()`	Liest den Rest der Zeile.
`static`	`String`	`readAll()`	Liest den Rest des Eingabestroms.

Umleitung und Pipelining

Standardeingabe und -ausgabe erlauben uns, Befehlszeilenerweiterungen zu nutzen, die von vielen Betriebssystemen unterstützt werden. Durch eine einfache zusätzliche Direktive im Programmaufruf können wir die Standardausgabe des Programms in eine Datei *umleiten*, um sie dort permanent zu speichern oder später als Eingabe für ein anderes Programm zu verwenden. Der Befehl

```
% java RandomSeq 1000 100.0 200.0 > data.txt
```

zum Beispiel gibt an, dass der Standardausgabestrom nicht im Konsolenfenster ausgegeben, sondern in eine Textdatei namens *data.txt* geschrieben werden soll. Jeder Aufruf von `System.out.print()` oder `System.out.println()` hängt Text an das Ende dieser Datei an. In diesem Fall ist das Endergebnis eine Datei, die 1000 Zufallswerte enthält. Im Konsolenfenster erscheint keine Ausgabe; alle Ausgaben werden direkt in die Datei umgeleitet, die hinter dem >-Symbol angegeben ist. Auf diese Weise können wir Informationen für eine spätere Verwendung speichern. Beachten Sie, dass wir keinerlei Änderungen an `RandomSeq` vornehmen müssen, damit dieser Mechanismus funktioniert – `RandomSeq` verwendet einfach die Standardausgabeabstraktion und bleibt davon unberührt, dass wir eine andere Implementierung dieser Abstraktion verwenden. Auf gleiche Weise können wir auch die Standardeingabe umleiten, sodass `StdIn` die Daten aus einer Datei und nicht von der Konsolenanwendung liest:

```
% java Average < data.txt
```

Dieser Befehl liest eine Folge von Zahlen aus der Datei *data.txt* und berechnet deren Mittelwert. Hierbei ist das Symbol < eine Direktive, die das Betriebssystem anweist, den Standardeingabestrom aus dem Inhalt der Textdatei *data.txt* aufzubauen, anstatt darauf

zu warten, dass der Benutzer irgendetwas in das Konsolenfenster eingibt. Wenn das Programm `StdIn.readDouble()` aufruft, liest das Betriebssystem den Wert aus der Datei ein. Die Kombination beider Mechanismen, mit dem Ziel die Ausgabe eines Programms so umzuleiten, dass sie zur Eingabe eines anderen wird, bezeichnen wir als *Pipelining*:

```
% java RandomSeq 1000 100.0 200.0 | java Average
```

Dieser Befehl gibt zum Beispiel an, dass die Standardausgabe von `RandomSeq` und die Standardeingabe von `Average` *derselbe* Strom sind – so als würde `RandomSeq` die von ihm erzeugten Zahlen direkt in das Konsolenfenster des gerade ausgeführten Programms `Average` eingeben. Dies ist ein gravierender Unterschied zu dem zuvor geschilderten Verfahren, da es uns von der Beschränkung hinsichtlich der Größe der von uns verarbeiteten Ein- und Ausgabeströme befreit. Wir könnten beispielsweise den Wert `1000` in unserem Beispiel durch `1000000000` ersetzen, selbst wenn wir nicht genug Platz haben, um eine Milliarde Zahlen auf unserem Computer zu speichern (allerdings benötigen wir die *Rechenzeit*, um sie zu verarbeiten). Wenn `RandomSeq` die Methode `StdOut.println()` aufruft, wird ein String an das Ende des Stroms angehängt, und wenn `Average` dann `StdIn.readInt()` aufruft, wird ein String vom Anfang des Stroms entfernt. Für die genaue zeitliche Abstimmung ist dabei das Betriebssystem zuständig: Es führt unter Umständen `RandomSeq` aus, bis es eine Ausgabe erzeugt, und führt dann `Average` aus, um diese Ausgabe weiterzuverarbeiten; oder es führt `Average` aus, bis dieses Programm eine Eingabe benötigt, und führt dann `RandomSeq` aus, bis es die benötigte Ausgabe erzeugt. Das Endergebnis ist das gleiche, aber unsere Programme müssen sich jetzt um solche Details keine Gedanken mehr machen, denn sie arbeiten allein mit den Abstraktionen der Standardeingabe und -ausgabe.

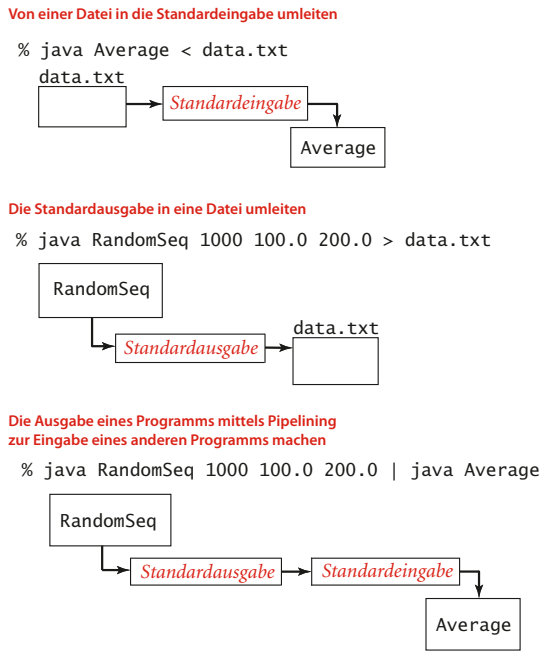


Abbildung 1.7: Umleitung und Pipelining von der Befehlszeile aus

Eingabe und Ausgabe aus einer Datei

Unsere Bibliotheken `In` und `Out` bieten statische Methoden zur Implementierung der Abstraktion, den Inhalt eines Arrays von Werten eines primitiven Datentyps (oder `String`) aus einer Datei zu lesen und in eine Datei zu schreiben. Wir verwenden die Methoden `readInts()`, `readDoubles()` und `readStrings()` der Bibliothek `In` und die Methoden `writeInts()`, `writeDoubles()` und `writeStrings()` der Bibliothek `Out`. Das benannte Argument kann eine Datei oder eine Webseite sein. Dieser Mechanismus erlaubt uns beispielsweise, eine Datei und die Standardeingabe für zwei verschiedene Zwecke im selben Programm zu verwenden (wie in `BinarySearch`). Die Bibliotheken `In` und `Out` implementieren darüber hinaus Datentypen mit Instanzmethoden, die es erlauben, Dateien als Ein- und Ausgabeströme und Webseiten als Eingabeströme zu behandeln. Aus diesem Grund werden wir in Abschnitt 1.2 noch einmal darauf zu sprechen kommen.

Tabelle 1.18 APIs für unsere statischen Methoden zum Lesen und Schreiben von Arrays

public class In

`static`	`int[]`	`readInts(String name)`	Liest `int`-Werte.
`static`	`double[]`	`readDoubles(String name)`	Liest `double`-Werte.
`static`	`String[]`	`readString(String name)`	Liest `String`-Werte.

public class Out

`static`	`void`	`write(int[] a, String name)`	Schreibt `int`-Werte.
`static`	`void`	`write(double[] a, String name)`	Schreibt `double`-Werte.
`static`	`void`	`write(String[] a, String name)`	Schreibt `String`-Werte.

Hinweis 1: Es werden auch andere primitive Datentypen unterstützt.
Hinweis 2: `StdIn` und `StdOut` werden unterstützt (Argument `name` weglassen).

Standardgrafik (grundlegende Methoden)

Bisher haben sich unsere Eingabe-/Ausgabeabstraktionen ausschließlich auf Textstrings beschränkt. Jetzt führen wir eine Abstraktion ein, die als Ausgabe Grafiken produziert. Die zugrunde liegende Bibliothek ist einfach und ermöglicht uns, von einem visuellen Medium Gebrauch zu machen, sodass wir mit wesentlich mehr Informationen zurecht kommen, als wenn wir nur Text hätten. Wie im Fall der Standardeingabe ist unsere Standardgrafik-Abstraktion in einer eigenen Bibliothek namens `StdDraw` implementiert (*StdDraw.java*), die Sie von der Website zum Buch in Ihr Arbeitsverzeichnis herunterladen müssen. Standardgrafik ist von der Konzeption her sehr einfach: Stellen Sie sich ein simples abstraktes Zeichengerät vor, das Linien und Punkte auf eine zweidimensionale Leinwand zeichnen kann und in der Lage ist, auf Befehle zu reagieren, die von unseren Programmen in Form von Aufrufen statischer `StdDraw`-Methoden an das Ge rät gesendet werden. Hierzu gehören Methoden zum Zeichnen von Linien, Punkten,

Textstrings, Kreisen, Rechtecken und Polygonen. Wie die Methoden für die Standardeingabe und -ausgabe sind diese Methoden fast selbsterklärend: `StdDraw.line()` zeichnet ein gerades Liniensegment, das die Punkte (x_0, y_0) und (x_1, y_1) miteinander verbindet. Die Koordinaten der Punkte werden als Argumente übergeben. `StdDraw.point()` zeichnet einen Punkt an der Stelle (x, y). Die Koordinaten des Punkts werden auch hier als Argumente übergeben. (Siehe hierzu auch die Diagramme in ▶Abbildung 1.8.) Die zugrunde liegende Standardskala ist das Einheitsquadrat (alle Koordinaten zwischen 0 und 1). Die Standardimplementierung zeigt die Leinwand in einem Fenster auf Ihrem Bildschirm, mit schwarzen Linien und Punkten auf einem weißen Hintergrund.

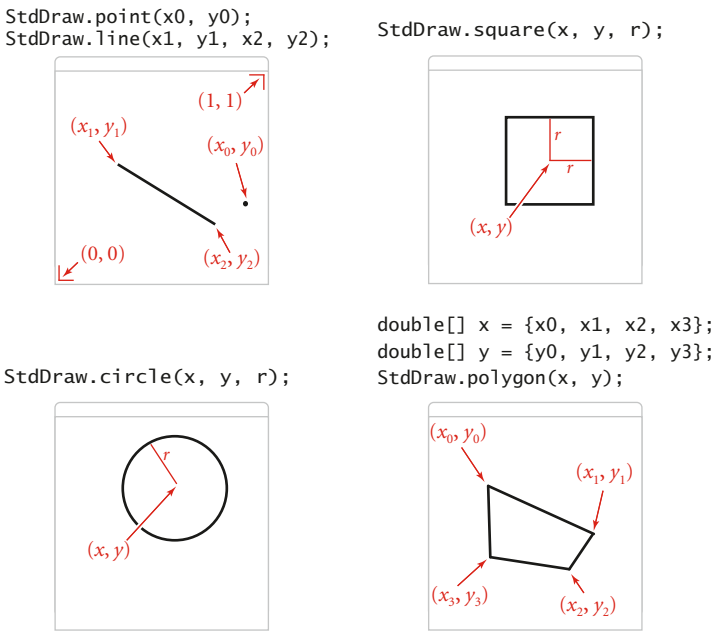


Abbildung 1.8: Beispiele für `StdDraw`

Tabelle 1.19 API für unsere Bibliothek der statischen Standardgrafik-Methoden

public class StdDraw

static	void	line(double x0, double y0, double x1, double y1)
static	void	point(double x, double y)
static	void	text(double x, double y, String s)
static	void	circle(double x, double y, double r)
static	void	filledCircle(double x, double y, double r)
static	void	ellipse(double x, double y, double rw, double rh)
static	void	filledEllipse(double x, double y, double rw, double rh)

Tabelle 1.19 API für unsere Bibliothek der statischen Standardgrafik-Methoden *(Forts.)*

public class StdDraw

static	void	square(double x, double y, double r)
static	void	filledSquare(double x, double y, double r)
static	void	rectangle(double x, double y, double rw, double rh)
static	void	filledRectangle(double x, double y, double rw, double rh)
static	void	polygon(double[] x, double[] y)
static	void	filledPolygon(double[] x, double[] y)

Standardgrafik (Konfigurationsmethoden)

In der Bibliothek finden Sie auch Methoden, mit denen Sie die Skalierung und Größe der Leinwand, die Farbe und Stärke der Linien, die Schriftart und den Zeitpunkt der Zeichenaktion (wichtig für Animationen) ändern können. Als Argumente für `setPenColor()` stehen Ihnen vordefinierte Farben wie `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `BOOK_RED`, `WHITE` und `YELLOW` zur Verfügung, die als Konstanten in `StdDraw` definiert sind. (So schreiben wir beispielsweise `StdDraw.RED`, um auf eine der Farben Bezug zu nehmen.) Das Fenster verfügt außerdem über eine Menüoption, mit der Sie Ihre Grafik in einer Datei speichern können, und zwar in einem Format, das die Veröffentlichung im Web erlaubt.

Tabelle 1.20 API für unsere Bibliothek der statischen Standardgrafik-Methoden (Konfigurationsmethoden)

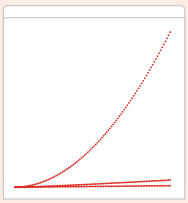
public class StdDraw

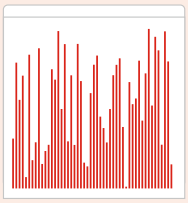
static	void	setXscale(double x0, double x1)	Setzt den x-Bereich auf (x_0, x_1) zurück.
static	void	setYscale(double y0, double y1)	Setzt den y-Bereich auf (y_0, y_1) zurück.
static	void	setPenRadius(double r)	Setzt den Stiftradius auf `r`.
static	void	setPenColor(Color c)	Setzt die Stiftfarbe auf `c`.
static	void	setFont(Font f)	Setzt die Schriftart auf `f`.
static	void	setCavasSize(int w, int h)	Setzt die Leinwand auf ein `w`×`h`-Fenster.
static	void	clear(Color c)	Löscht die Leinwand und färbt sie mit der Farbe `c`.
static	void	show(int dt)	Zeigt alles; pausiert `dt` Millisekunden.

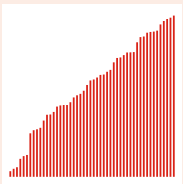
Wir verwenden `StdDraw` in diesem Buch für die Datenanalyse und die Erstellung visueller Repräsentationen von Algorithmen, die sich in der Ausführung befinden. Die nachfolgende ▶Tabelle 1.21 zeigt einige Beispiele; weitere werden wir im weiteren Verlauf des Buches und in den Übungen betrachten. Die Bibliothek unterstützt darüber hinaus *Animationen* – ein Thema, das hauptsächlich auf der Website zum Buch behandelt wird.

Tabelle 1.21 `StdDraw`-Beispiele zum Plotten

Daten	Plotimplementierung (Codefragment)	Ergebnis

Funktionswerte	`int N = 100;`
	`StdDraw.setXscale(0, N);`
	`StdDraw.setYscale(0, N*N);`
	`StdDraw.setPenRadius(.01);`
	`for (int i = 1; i <= N; i++)`
	`{`
	`  StdDraw.point(i, i);`
	`  StdDraw.point(i, i*i);`
	`  StdDraw.point(i, i*Math.log(i));`
	`}`
		

Array von Zufallswerten	`int N = 50;`
	`double[] a = new double[N];`
	`for (int i = 0; i < N; i++)`
	`  a[i] = StdRandom.random();`
	`for (int i = 0; i < N; i++)`
	`{`
	`  double x = 1.0*i/N;`
	`  double y = a[i]/2.0;`
	`  double rw = 0.25/N;`
	`  double rh = a[i]/2.0;`
	`  StdDraw.filledRectangle(x, y, rw, rh);`
	`}`
		

Sortiertes Array von Zufallswerten	`int N = 50;`
	`double[] a = new double[N];`
	`for (int i = 0; i < N; i++)`
	`  a[i] = StdRandom.random();`
	`Arrays.sort(a);`
	`for (int i = 0; i < N; i++)`
	`{`
	`  double x = 1.0*i/N;`
	`  double y = a[i]/2.0;`
	`  double rw = 0.25/N;`
	`  double rh = a[i]/2.0;`
	`  StdDraw.filledRectangle(x, y, rw, rh);`
	`}`
		

1.1.10 Binäre Suche

Das Java-Programm, das wir Ihnen als Erstes präsentiert haben, basiert auf einem berühmten, effizienten und häufig anzutreffenden Algorithmus: der *binären Suche*. Dieses Beispiel veranschaulicht exemplarisch, wie wir im weiteren Verlauf des Buches neue Algorithmen untersuchen werden. Wie alle Programme, die wir betrachten, bietet es zweierlei: eine genaue Definition des umgesetzten Verfahrens und eine vollständige Java-Implementierung, die Sie von der Website zum Buch herunterladen können.

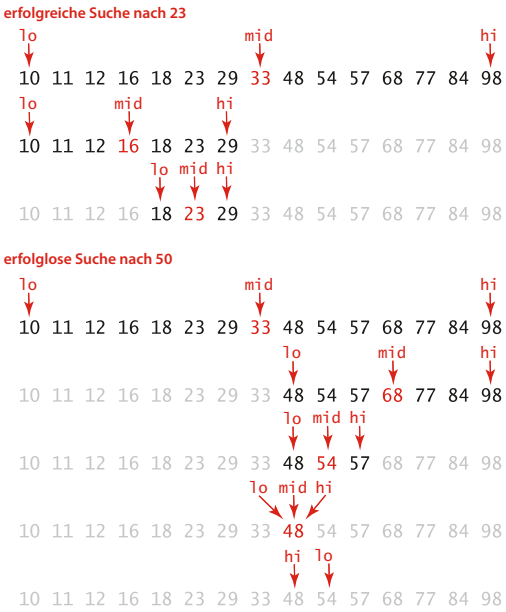


Abbildung 1.9: Binäre Suche in einem geordneten Array

Binäre Suche

Wir werden den Algorithmus der binären Suche zwar erst in *Kapitel 3.2* ausführlich behandeln, doch wollen wir Ihnen hier bereits eine kurze Beschreibung geben. Der Algorithmus ist in der statischen Methode `rank()` implementiert, die einen Integerschlüssel und ein *sortiertes* Array von `int`-Werten als Argumente übernimmt und entweder den Index des Schlüssels zurückliefert, wenn der Schlüssel im Array vorhanden ist, oder `-1`. Hierzu führen wir die beiden Variablen `lo` und `hi` ein, sodass der Schlüssel im Bereich `a[lo..hi]` liegt, wenn er sich im Array befindet. Anschließend treten wir in eine Schleife ein, die den mittleren Eintrag im Intervall prüft (an Indexposition `mid`). Wenn der Schlüssel gleich `a[mid]` ist, lautet der Rückgabewert `mid`; andernfalls halbiert die Methode die Intervallgröße und sucht in der linken Hälfte, wenn der Schlüssel kleiner als `a[mid]` ist, bzw. in der rechten, wenn der Schlüssel größer als `a[mid]` ist. Der Prozess endet, wenn der Schlüssel gefunden wurde oder das Intervall leer ist. Die binäre Suche ist äußerst effizient, da nur ein paar Arrayelemente (relativ zur Größe des Arrays) untersucht werden müssen, um den Schlüssel zu finden (oder festzustellen, dass er nicht vorhanden ist).

Entwicklungsclient

Zu jeder Algorithmusimplementierung liefern wir einen Entwicklungsclient `main()`, mit dem Sie den Algorithmus auf Herz und Nieren prüfen und seine Laufzeit testen können. Verwenden Sie dazu die Eingabedateien aus dem Buch und von der Website zum Buch. In unserem Beispiel liest der Client ganzzahlige Werte aus einer Datei, die auf der Befehlszeile angegeben wird, und gibt dann auf der Standardeingabe alle ganzzahligen Werte aus, die nicht in der Datei stehen. Zur Veranschaulichung dieses Verhaltens verwenden wir kleine Testdateien wie in ▶Abbildung 1.10, die auch als Grundlage für Ablaufprotokolle und Beispiele wie in Abbildung 1.9 dienen. Zur Modellierung realer Anwendungen und für Laufzeittests verwenden wir normalerweise sehr große Testdateien (siehe Abschnitt *Positivlisten*).

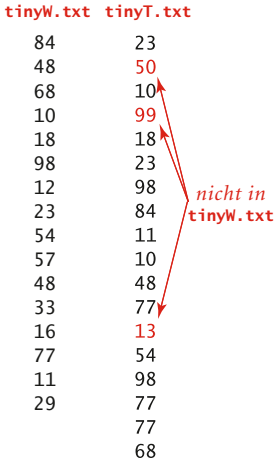


Abbildung 1.10: Kleine Testdateien für den Testclient `BinarySearch`

Positivlisten

Unsere Entwicklungsclients sind möglichst so ausgelegt, dass sie praktische Situationen widerspiegeln und die Notwendigkeit des vorliegenden Algorithmus demonstrieren. In diesem Fall geht es um ein Verfahren, das auch als *Whitelisting* bezeichnet wird. Stellen Sie sich beispielsweise ein Kreditkartenunternehmen vor, das prüfen muss, ob die Kundentransaktionen einem gültigen Konto zuzuordnen sind. Dazu kann das Unternehmen

- die Kontonummern der Kunden in einer Datei speichern, die wir als *Positivliste* oder auch *Weiße Liste* (*whitelist*) bezeichnen,
- über den Standardeingabestrom die Kontonummern einlesen, die mit den jeweiligen Transaktionen verbunden sind,
- mithilfe des Testclients die Zahlen auf der Standardausgabe ausgeben, die *nicht* mit einem Kunden verbunden sind. Diese Transaktionen würde das Unternehmen wahrscheinlich ablehnen.

Es ist nicht ungewöhnlich, dass ein großes Unternehmen Millionen von Kunden hat und Millionen von Transaktionen oder mehr verarbeiten muss. Damit Sie diese Situation

nachvollziehen können, finden Sie auf der Website zum Buch die Dateien *largeW.txt* (1 Million ganze Zahlen) und *largeT.txt* (10 Millionen ganze Zahlen).

Listing 1.4: Binäre Suche

```
import java.util.Arrays;
public class BinarySearch
{
   public static int rank(int key, int[] a)
   {  // Array muss sortiert sein.
      int lo  = 0;
      int hi = a.length - 1;
      while (lo <= hi)
      {  // Schlüssel key ist in a[lo..hi] oder nicht vorhanden.
         int mid = lo + (hi - lo) / 2;
         if      (key < a[mid]) hi = mid - 1;
         else if (key > a[mid]) lo = mid + 1;
         else                   return mid;
      }
      return -1;
   }
   public static void main(String[] args)
   {
      int[] whitelist = In.readInts(args[0]);
      Arrays.sort(whitelist);
      while (!StdIn.isEmpty())
      {  // Liest Schlüssel ein, gibt ihn aus, wenn er nicht in der
         // Positivliste ist.
         int key = StdIn.readInt();
         if (rank(key, whitelist) < 0)
            StdOut.println(key);
      }
   }
}
```

Dieses Programm übernimmt als Argument den Dateinamen einer Positivliste (eine Folge von ganzen Zahlen) und filtert dann alle Einträge aus der Standardeingabe heraus, die in der Positivliste stehen, sodass in der Standardausgabe nur ganze Zahlen erscheinen, die nicht in der Positivliste stehen. Hierzu verwendet das Programm den Algorithmus zur binären Suche, der in der statischen Methode `rank()` implementiert ist und die Aufgabe effizient löst. In Kapitel 3.1 werden der Algorithmus zur binären Suche, seine Korrektheit, die Laufzeitanalyse und seine Anwendungen ausführlich diskutiert.

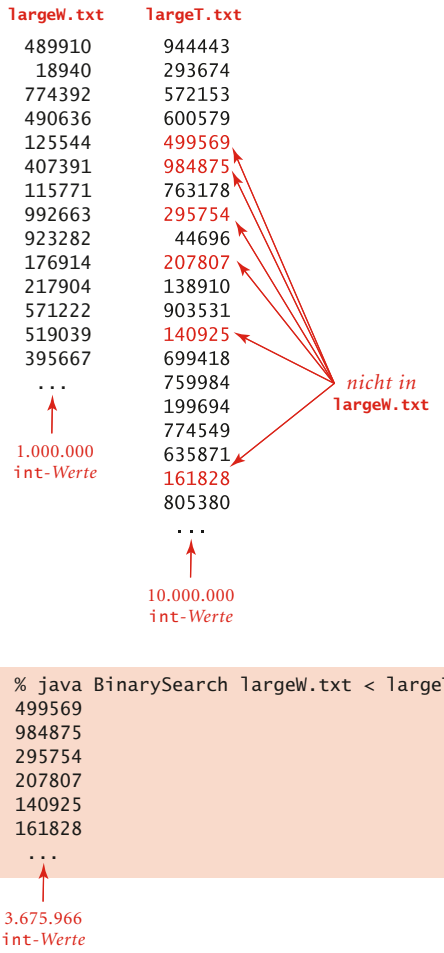
```
% java BinarySearch tinyW.txt < tinyT.txt
50
99
13
```

Performance

Oft reicht es nicht, wenn ein Programm einfach funktioniert. So gibt es beispielsweise eine viel einfachere Implementierung von `rank()`, die nicht einmal voraussetzt, dass das Array sortiert ist, um jeden Eintrag zu prüfen:

```
public static int rank(int key, int[] a)
{
   for (int i = 0; i < a.length; i++)
      if (a[i] == key) return i;
   return -1;
}
```

Diese einfache und leicht nachzuvollziehende Lösung wirft die Frage auf, wozu wir Mergesort und die binäre Suche benötigen? Spätestens bei *Übung 1.1.38* werden Sie feststellen, dass Ihr Computer zu langsam ist, um mit dieser Brute-Force-Implementierung von `rank()` große Eingabemengen (z.B. 1 Million Einträge in der Positivliste und 10 Millionen Transaktionen) zu verarbeiten. *Das Positivlisten-Problem bei einer großen Anzahl von Eingaben lässt sich nur mit effizienten Algorithmen wie der binären Suche und Mergesort lösen*. Ein gutes Laufzeitverhalten ist oft von entscheidender Bedeutung, sodass wir hier den Grundstein legen, um die Laufzeit in Abschnitt 1.4 zu studieren und das Laufzeitverhalten aller unserer Algorithmen zu analysieren (einschließlich der binären Suche in *Kapitel 3.1* und Mergesort in *Kapitel 2.2*).




```
% java BinarySearch largeW.txt < largeT.txt
499569
984875
295754
207807
140925
161828
 ...
```

3.675.966 int-Werte

Abbildung 1.11: Große Dateien für den `BinarySearch`-Testclient

Mit der ausführlichen Darstellung unseres Programmiermodells wollen wir sicherstellen, dass Sie auf Ihrem Computer Code wie `BinarySearch` auf unseren Testdaten ausführen und an verschiedene Situationen anpassen können (wie jene, die in den Übungen am Ende dieses Abschnitts beschrieben werden), um sich von der Anwendbarkeit des Modells zu überzeugen. Das hier geschilderte Programmiermodell wurde entwickelt, um speziell Aktivitäten wie diese zu erleichtern, die für unseren Ansatz, Algorithmen zu studieren, wichtig sind.

1.1.11 Ausblick

In diesem Abschnitt haben wir ein genaues und vollständiges Programmiermodell beschrieben, das vielen Programmierern seit Jahrzehnten gute Dienste geleistet hat (und immer noch leistet). Die moderne Programmierung geht jedoch noch einen Schritt weiter. Diese nächste Ebene wird *Datenabstraktion* (manchmal auch *objektorientierte Programmierung*) genannt und ist Thema des nächsten Abschnitts. Einfach ausgedrückt verbirgt sich dahinter die Idee, dass ein Programm selbst *Datentypen* (Sätze von Werten und Sätze von Operationen auf diesen Werten) definieren kann und nicht nur statische Methoden, die auf vordefinierten Datentypen operieren.

Die objektorientierte Programmierung erfreut sich in den letzten Jahrzehnten einer wachsenden Beliebtheit und Datenabstraktion ist das A und O der modernen Programmentwicklung. Wir widmen der Datenabstraktion in diesem Kapitel aus drei Gründen einen eigenen Abschnitt:

- Dank modularer Programmierung können wir Code in viel größerem Maße wiederverwenden. So erlauben es unsere Sortieralgorithmen aus *Kapitel 2* sowie die binäre Suche und andere Algorithmen aus *Kapitel 3* den Clients, den gleichen Code für jeden beliebigen Datentyp (nicht nur Integer) zu verwenden – selbst für Datentypen, die vom Client definiert wurden.
- Datenabstraktion bietet einen bequemen Mechanismus, um sogenannte *verkettete* Datenstrukturen zu erstellen, die flexibler sind als Arrays und nicht selten die Grundlage effizienter Algorithmen bilden.
- Datenabstraktion ermöglicht uns, die algorithmischen Herausforderungen, vor denen wir stehen, genau zu definieren. So legen wir beispielsweise in unseren Union-Find-Algorithmen in Abschnitt 1.5, unseren Algorithmen zu Prioritätswarteschlangen in *Abschnitt 2.4* und unseren Symboltabellenalgorithmen in *Kapitel 3* das Hauptaugenmerk auf die Definition von Datenstrukturen, die effiziente Implementierungen einer *Menge* von Operationen erlauben. Diese Herausforderung deckt sich perfekt mit der Datenabstraktion.

Trotz dieser Überlegungen liegt unser Fokus natürlich weiterhin auf dem Studium der Algorithmen, weswegen wir im Folgenden, wenn wir uns näher mit der objektorientierten Programmierung beschäftigen, das Hauptaugenmerk auf Konzepte legen werden, die für unsere Zwecke von Bedeutung sind.

1.1 Fragen und Antworten

Frage: Was ist Java-Bytecode?

Antwort: Eine maschinennahe Version Ihres Programms, die auf der *Java Virtual Machine* läuft. Diese Abstraktionsebene erleichtert es den Java-Entwicklern sicherzustellen, dass unsere Programme auf einer Vielzahl verschiedener Geräte ausgeführt werden können.

Frage: Ich finde es nicht richtig, dass Java bei `int`-Werten einfach einen Überlauf erzeugt und fehlerhafte Werte liefert. Sollte Java nicht automatisch auf Überlauf prüfen?

Antwort: Dieses Thema ist unter Programmierern sehr umstritten. Um die Antwort im Moment noch kurz zu halten, sei gesagt, dass die fehlende Prüfung der Grund ist, warum wir diese Typen als *primitive* Datentypen bezeichnen. Ein wenig Hintergrundwissen in diesem Bereich kann viel dazu beitragen, Fehler wie diese zu vermeiden. Allgemein gilt, dass Sie für kleine Zahlen (weniger als 10 Ziffern) problemlos den Datentyp `int` verwenden können, doch wenn die Werte in die Milliarden gehen, müssen Sie den Typ `long` wählen.

Frage: Wie lautet der Wert von `Math.abs(-2147483648)`?

Antwort: `-2147483648`. Dieses befremdliche (aber korrekte) Ergebnis ist ein typisches Beispiel für die Auswirkungen von Integer-Überläufen.

Frage: Wie kann ich eine `double`-Variable mit Unendlich initialisieren?

Antwort: Java stellt Ihnen hierfür vordefinierte Konstanten zur Verfügung: `Double.POSITIVE_INFINITY` und `Double.NEGATIVE_INFINITY`.

Frage: Kann man einen `double`-Wert mit einem `int`-Wert vergleichen?

Antwort: Nicht ohne eine Typumwandlung. Denken Sie jedoch daran, dass Java in der Regel die notwendige Umwandlung automatisch vornimmt. Wenn zum Beispiel `x` ein `int` mit dem Wert `3` ist, dann ist der Ausdruck (`x < 3.1`) wahr; Java wandelt `x` in einen `double`-Wert um (da `3.1` ein `double`-Literal ist), bevor der Vergleich durchgeführt wird.

Frage: Was passiert, wenn ich eine Variable verwende, bevor ich sie mit einem Wert initialisiert habe?

Antwort: Java meldet einen Kompilierfehler, wenn es einen Pfad durch Ihren Code gibt, der zur Verwendung einer nicht initialisierten Variablen führen könnte.

Frage: Welchen Wert haben `1/0` und `1.0/0.0` als Java-Ausdrücke?

Antwort: Der erste Ausdruck erzeugt wegen der Division durch null zur Laufzeit eine *Ausnahme* (woraufhin Ihr Programm anhält, weil der Wert undefiniert ist). Der zweite Ausdruck ist zulässig und hat den Wert `Infinity`.

Frage: Kann man die Operatoren `<` und `>` verwenden, um `String`-Variablen zu vergleichen?

Antwort: Nein. Diese Operatoren sind nur für primitive Datentypen definiert worden. (▶Tabelle 1.29.)

Frage: Was ist das Ergebnis von Divisions- und Modulo-Berechnungen mit negativen Integern?

Antwort: Der Quotient von `a / b` wird abgerundet; der Rest von `a % b` ist so definiert, dass `(a/b) * b + a % b` immer gleich `a` ist. So sind zum Beispiel `-14/3` und `14/-3` beide `-4`, aber `-14 % 3` ist `-2` und `14 % -3` ist `2`.

Frage: Warum sagen wir `(a && b)` und nicht `(a & b)`?

Antwort: Die Operatoren `&`, `|` und `^` gehören zu den *bitweisen* logischen Operationen für Integer-Typen, die auf jede Bitposition *Und*-, *Oder*- oder *Xor*-Operationen ausführen. Demzufolge ist der Wert von `10|6` gleich `14` und der Wert von `10^6` gleich `12`. Wir verwenden diese Operatoren in diesem Buch nur selten. Die nur für die booleschen Ausdrücke gültigen Operatoren `&&` und `||` wurden gesondert aufgenommen und sind *Kurzschlussoperatoren*: Ein Ausdruck wird von links nach rechts ausgewertet und die Auswertung ist beendet, sobald der logische Wert bekannt ist.

Frage: Ist Doppeldeutigkeit in verschachtelten `if`-Anweisungen ein Problem?

Antwort: Ja. Wenn Sie in Java schreiben

```
if <Ausdr1> if <Ausdr2> <AnwsgA> else <AnwsgB>
```

entspricht dies dem Code

```
if <Ausdr1> if { <Ausdr2> <AnwsgA> else <AnwsgB> }
```

auch wenn Sie vielleicht gedacht hätten, es hieße

```
if <Ausdr1> { if <Ausdr2> <AnwsgA> } else <AnwsgB>
```

Auch hier gilt, dass geschweifte Klammern erheblich dazu beitragen können, Fehler dieser Art zu vermeiden.

Frage: Was ist der Unterschied zwischen einer `for`-Schleife und ihrer `while`-Formulierung?

Antwort: Der Code im Kopf der `for`-Schleife wird dem gleichen Block zugerechnet wie der Rumpf der `for`-Schleife, weswegen in einer typischen `for`-Schleife die Inkrementvariable – anders als in einer äquivalenten `while`-Schleife – nicht für weitere Anweisungen zur Verfügung steht. Dies ist oft ein Grund, eine `while`-Schleife statt einer `for`-Schleife zu verwenden.

Frage: Einige Java-Programmierer verwenden `int a[]` anstelle von `int[] a`, um Arrays zu deklarieren. Wo liegt der Unterschied?

Antwort: In Java sind beide Schreibweisen äquivalent und zulässig. Die erste Schreibweise entspricht der Arraydeklaration in C, während letztere der bevorzugte Stil in Java ist, da der Typ der Variablen `int[]` deutlicher anzeigt, dass es sich dabei um ein *Array* von Integerwerten handelt.

Frage: Warum beginnen Arrayindizes mit `0` anstatt mit `1`?

Antwort: Diese Konvention stammt noch aus der Zeit der maschinennahen Programmierung, als die Adresse eines Arrayelements berechnet wurde, indem der Index zur Adresse des Arrayanfangs addiert wurde. Hätte man die Indizes mit 1 begonnen, wäre damit entweder eine Menge Platz am Anfang des Arrays oder eine Menge Zeit für die Subtraktion der 1 verschwendet worden.

Frage: Wenn `a[]` ein Array ist, warum gibt `StdOut.println(a)` einen hexadezimalen Wert wie `@f62373` aus anstelle der Elemente des Arrays?

Antwort: Gute Frage. Der Befehl gibt die Speicheradresse des Arrays aus, was wohl in den seltensten Fällen gewünscht ist.

Frage: Warum verwenden wir nicht die Standardbibliotheken von Java für Eingabe und Grafik?

Antwort: Wir *verwenden sie ja*, aber wir ziehen es vor, mit einfacheren abstrakten Modellen zu arbeiten. Die Java-Bibliotheken hinter `StdIn`, `StdDraw` und `StdAudio` sind auf die produktive Programmierung ausgerichtet, weswegen die Bibliotheken und ihre APIs etwas unhandlich sind. Wenn Sie sich einen Eindruck davon verschaffen wollen, werfen Sie einen Blick in den Code von *StdIn.java* und *StdDraw.java*.

Frage: Kann mein Programm Daten von der Standardeingabe erneut einlesen?

Antwort: Nein. Sie haben nur einen Versuch. Sie können ja auch einen `println()`-Befehl nicht zurücknehmen.

Frage: Was passiert, wenn mein Programm versucht, Daten aus einer Standardeingabe zu lesen, die leer ist?

Antwort: Sie erhalten einen Fehler. Die Methode `StdIn.isEmpty()` ermöglicht Ihnen, solche Fehler zu vermeiden, indem sie prüft, ob noch weitere Eingaben verfügbar sind.

Frage: Was bedeutet die Fehlermeldung: `Exception in thread "main" java.lang.NoClassDefFoundError: StdIn`?

Antwort: Sie haben wahrscheinlich vergessen, *StdIn.java* in Ihrem Arbeitsverzeichnis abzulegen.

Frage: Kann eine statische Methode in Java eine andere statische Methode als Argument übernehmen?

Antwort: Nein. Gute Frage, da viele andere Programmiersprachen diese Möglichkeit unterstützen.

1.1 Allgemeine Übungen

1. Wie lautet jeweils der Wert der folgenden Ausdrücke?

a. `(0 + 15) / 2`

b. `2.0e-6 * 100000000.1`

c. `true && false || true && true`

2. Wie lauten jeweils der Typ und der Wert der folgenden Ausdrücke?

a. `(1 + 2.236) / 2`

b. `1 + 2 + 3 + 4.0`

c. `4.1 >= 4`

d. `1 + 2 + "3"`

3. Schreiben Sie ein Programm, das drei Integer als Befehlszeilenargumente übernimmt und `gleich` ausgibt, wenn alle drei gleich sind, beziehungsweise ansonsten `nicht gleich`.

4. Was (wenn überhaupt) ist falsch an den folgenden Anweisungen?

a. `if (a > b) then c = 0;`

b. `if a > b { c = 0; }`

c. `if (a > b) c = 0;`

d. `if (a > b) c = 0 else b = 0;`

5. Schreiben Sie ein Codefragment, das `true` ausgibt, wenn die `double`-Variablen `x` und `y` beide zwischen `0` und `1` liegen, und im anderen Fall `false`.

6. Was gibt das folgende Programm aus?

```
int f = 0;
int g = 1;
for (int i = 0; i <= 15; i++)
{
   StdOut.println(f);
   f = f + g;
   g = f - g;
}
```

7. Wie lautet der Wert, der von jedem der folgenden Codefragmente ausgegeben wird?

a.
```
double t = 9.0;
while (Math.abs(t - 9.0/t) > .001)
   t = (9.0/t + t) /2.0;
StdOut.printf("%.5f\n", t);
```

b.
```
int sum = 0;
for (int i = 1; i < 1000; i++)
   for (int j = 0; j < i; j++)
      sum++;
StdOut.println(sum);
```

c.
```
int sum = 0;
for (int i = 1; i < 1000; i *= 2)
   for (int j = 0; j < 1000; j++)
      sum++;
StdOut.println(sum);
```

8. Wie lautet die Ausgabe der folgenden Anweisungen?

a. `System.out.println('b');`

b. `System.out.println('b' + 'c');`

c. `System.out.println((char) ('a' + 4));`

Erläutern Sie die jeweiligen Ergebnisse.

9. Schreiben Sie ein Codefragment, das die binäre Darstellung einer positiven ganzen Zahl `N` in einem String `s` ablegt.

Lösung: Java stellt hierfür die Methode `Integer.toBinaryString(N)` zur Verfügung, aber Sinn und Zweck der Übung ist es natürlich zu sehen, wie eine solche Methode implementiert werden könnte. Eine besonders kurze Lösung könnte folgendermaßen aussehen:

```
String s = "";
for (int n = N; n > 0; n /= 2)
   s = (n % 2) + s;
```

10. Was ist falsch an dem folgenden Codefragment?

```
int[] a;
for (int i = 0; i < 10; i++)
   a[i] = i * i;
```

Lösung: Es weist `a[]` keinen Speicher mit `new` zu. Dieser Code führt zu einem Kompilierfehler der Sorte *variable a might not have been initialized*.

11. Schreiben Sie ein Codefragment, das den Inhalt eines zweidimensionalen booleschen Arrays ausgibt, wobei `*` den Wert `true` und ein Leerzeichen den Wert `false` repräsentieren soll. Geben Sie Zeilen- und Spaltennummern mit aus.

12. Was gibt das folgende Codefragment aus?

```
int[] a = new int[10];
for (int i = 0; i < 10; i++)
   a[i] = 9 - i;
for (int i = 0; i < 10; i++)
   a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
   System.out.println(i);
```

13. Schreiben Sie ein Codefragment, das die Zeilen und Spalten eines zweidimensionalen Arrays mit *M* Zeilen und *N* Spalten vertauscht (*transponiert*) und ausgibt.

14. Schreiben Sie eine statische Methode `lg()`, die einen `int`-Wert `N` als Argument übernimmt und den größten Integer zurückliefert, der nicht größer ist als der Logarithmus von `N` zur Basis 2. Verwenden Sie *nicht* `Math`.

15. Schreiben Sie eine statische Methode `histogram()`, die ein Array `a[]` von `int`-Werten und einen Integer `M` als Argumente übernimmt und ein Array der Länge `M` zurückliefert, dessen `i`-ter Eintrag die Anzahl der Vorkommen des Integers `i` im Argumentarray angibt. Wenn die Werte in `a[]` alle zwischen `0` und `M-1` liegen, sollte die Summe der Werte in dem zurückgelieferten Array gleich `a.length` sein.

16. Wie lautet der Wert von `exR1(6)`:

```
public static String exR1(int n)
{
   if (n <= 0) return "";
   return exR1(n-3) + n + exR1(n-2) + n;
}
```

17. Analysieren Sie die folgende rekursive Funktion:

```
public static String exR2(int n)
{
   String s = exR2(n-3) + n + exR2(n-2) + n;
   if (n <= 0) return "";
   return s;

}
```

Antwort: Die Abbruchbedingung wird nie erreicht. Ein Aufruf von `exR2(3)` resultiert in Aufrufe von `exR2(0)`, `exR2(-3)`, `exR2(-6)` usw., bis ein Fehler vom Typ `StackOverflowError` auftritt.

18. Analysieren Sie die folgende rekursive Funktion:

```
public static int mystery(int a, int b)
{
   if (b == 0)     return 0;
   if (b % 2 == 0) return mystery(a+a, b/2);
   return mystery(a+a, b/2) + a;

}
```

Wie lauten die Werte von `mystery(2, 25)` und `mystery(3, 11)`? Beschreiben Sie, welchen Wert `mystery(a, b)` bei gegebenen positiven Integern `a` und `b` berechnet. Beantworten Sie die gleiche Frage, aber ersetzen Sie `+` durch `*` und `return 0` durch `return 1`.

19. Führen Sie folgendes Programm auf Ihrem Computer aus

```
public class Fibonacci
{
   public static long F(int N)
   {
      if (N == 0) return 0;
      if (N == 1) return 1;
      return F(N-1) + F(N-2);
   }
   public static void main(String[] args)
   {
      for (int N = 0; N < 100; N++)
```

```
            StdOut.println(N + " " + F(N));
        }
    }
```

Wie lautet der größte Wert von `N`, für den dieses Programm weniger als eine Stunde benötigt, um den Wert von `F(N)` zu berechnen? Entwickeln Sie eine bessere Implementierung von `F(N)`, die die berechneten Werte in einem Array speichert.

20. Schreiben Sie eine rekursive statische Methode, die den Wert von ln(*N*!) berechnet.

21. Schreiben Sie ein Programm, das Zeilen aus der Standardeingabe einliest, die jeweils einen Namen und zwei Integerwerte enthalten. Geben Sie anschließend mit `printf()` eine Tabelle aus, mit je einer Spalte für den Namen, die beiden Integer und das bis auf drei Dezimalstellen genaue Ergebnis der Division des ersten Integers durch den zweiten. Sie können ein solches Programm verwenden, um die mittleren Schlagleistungen von Baseballspielern oder die Noten von Schülern tabellarisch anzuordnen.

22. Schreiben Sie eine Version von `BinarySearch`, die die rekursive `rank()`-Methode aus Listing 1.1 verwendet und die Methodenaufrufe *nachverfolgt*. Jedes Mal, wenn die rekursive Methode aufgerufen wird, geben Sie die Argumentwerte `lo` und `hi` aus, eingerückt um die Tiefe der Rekursion. *Hinweis*: Ergänzen Sie die rekursive Methode um ein Argument, das die Rekursionstiefe aufnimmt.

23. Ergänzen Sie den `BinarySearch`-Testclient um die Fähigkeit, auf ein zweites Argument zu reagieren: `+`, um Zahlen von der Standardeingabe auszugeben, *die nicht in* der Positivliste sind, und `-`, um Zahlen auszugeben, *die in* der Positivliste sind.

24. Geben Sie die Abfolge der Werte von *p* und *q* an, die berechnet werden, wenn mit dem euklidischen Algorithmus der größte gemeinsame Teiler von `105` und `24` ermittelt wird. Erweitern Sie den Code aus Abbildung 1.1 zu einem Programm `Euclid`, das zwei Integer von der Befehlszeile entgegennimmt, ihren größten gemeinsamen Teiler berechnet und dabei die zwei Argumente für jeden Aufruf der rekursiven Methode ausgibt. Berechnen Sie mit Ihrem Programm den größten gemeinsamen Teiler von 1111111 und 1234567.

25. Beweisen Sie mittels vollständiger Induktion, dass der euklidische Algorithmus den größten gemeinsamen Teiler für ein beliebiges Paar nicht-negativer Integer *p* und *q* berechnet.

1.1 Knifflige Aufgaben

26. *Drei Zahlen sortieren*: Angenommen die Variablen `a`, `b`, `c` und `t` weisen alle den gleichen numerischen primitiven Datentyp auf. Beweisen Sie, dass der folgende Code `a`, `b` und `c` in aufsteigender Reihenfolge sortiert:

```
if (a > b) { t = a; a = b; b = t; }
if (a > c) { t = a; a = c; c = t; }
if (b > c) { t = b; b = c; c = t; }
```

27. *Binomialverteilung*: Schätzen Sie die Anzahl der rekursiven Aufrufe, die der folgende Code benötigen würde, um `binomial(100, 50, 0.2)` zu berechnen:

```
public static double binomial(int N, int k, double p)
{
   if (N == 0 || k == 0) return 1.0;
   if (N < 0 || k < 0) return 0.0;
   return (1.0 - p)*binomial(N-1, k, p) + p*binomial(N-1, k-1, p);
}
```

Entwickeln Sie eine bessere Implementierung, die darauf beruht, berechnete Werte in einem Array zu speichern.

28. *Duplikate entfernen*: Ändern Sie den Testclient in `BinarySearch` so, dass alle doppelten Schlüssel in der Positivliste nach dem Sortieren entfernt werden.

29. *Gleiche Schlüssel*: Ergänzen Sie `BinarySearch` um eine statische Methode `rank()`, die einen Schlüssel und ein sortiertes Array von `int`-Werten (von denen einige gleich sein können) entgegennimmt und die Anzahl der Elemente zurückliefert, die kleiner als der Schlüssel sind. Fügen Sie noch eine weitere ähnliche Methode namens `count()` hinzu, die die Anzahl der Elemente zurückliefert, die gleich dem Schlüssel sind. *Hinweis*: Wenn `i` und `j` die Werte sind, die von `rank(key, a)` und `count(key a)` zurückgeliefert werden, dann sind `a[i..i+j-1]` die Werte im Array, die gleich dem Schlüssel sind.

30. *Arrayübung*: Schreiben Sie ein Codefragment, das ein boolesches `N×N`-Array `a[][]` erzeugt, sodass `a[i][j]` `true` ist, wenn `i` und `j` relativ prim oder teilerfremd sind (d.h. keine gemeinsamen Faktoren haben), beziehungsweise im anderen Fall `false`.

31. *Zufallsverbindungen*: Schreiben Sie ein Programm, das als Befehlszeilenargumente einen Integer `N` und einen `double`-Wert `p` (zwischen 0 und 1) entgegennimmt, `N` Punkte der Größe 0,05 in gleichmäßigen Abständen auf den Umfang eines Kreises zeichnet und dann mit der Wahrscheinlichkeit `p` für jedes Punktepaar eine graue Verbindungslinie zwischen den Punkten zeichnet.

32. *Histogramm*: Angenommen, der Standardeingabestrom besteht aus einer Folge von `double`-Werten. Schreiben Sie ein Programm, das einen Integer *N* und zwei `double`-Werte *l* und *r* über die Befehlszeile entgegennimmt und unter Verwendung von `StdDraw` ein Histogramm erstellt, das den Bereich (*l*, *r*) in *N* gleich große

Intervalle teilt und dessen Balkenhöhen anzeigen, wie viele Zahlen aus der Standardeingabe jeweils in die einzelnen Intervalle fallen.

33. *Matrixbibliothek*: Schreiben Sie eine Bibliothek `Matrix`, die die folgende API implementiert:

public class Matrix

`static`	`double`	`dot(double[] x, double[] y)`	Skalarprodukt
`static`	`double[][]`	`mult(double[][] a, double[][] b)`	Matrizenprodukt
`static`	`double[][]`	`transpose(double[][] a)`	Transponierte
`static`	`double[]`	`mult(double[][] a, double[] x)`	Matrix-Vektor-Produkt
`static`	`double[]`	`mult(double[] y, double[] a)`	Vektor-Matrix-Produkt

Entwickeln Sie einen Testclient, der Werte von der Standardeingabe einliest und alle Methoden testet.

34. *Filtern*: Welche der folgenden Aufgaben *erfordern*, dass alle Werte aus der Standardeingabe gespeichert werden (zum Beispiel in einem Array), und welche könnten nur unter Verwendung einer festen Anzahl von Variablen und Arrays fester Größe (nicht abhängig von N) als Filter implementiert werden? Für jede Aufgabe kommt die Eingabe von der Standardeingabe und besteht aus N reellen Zahlen zwischen 0 und 1.

- Geben Sie den Maximal- und Minimalwert aus.
- Geben Sie den Median der Zahlen aus.
- Geben Sie den kt-kleinsten Wert aus (für k kleiner als 100).
- Geben Sie die Summe der Quadrate der Zahlen aus.
- Geben Sie den Mittelwert von N Zahlen aus.
- Geben Sie den Prozentsatz der Zahlen aus, die größer als der Mittelwert sind.
- Geben Sie die N Zahlen in aufsteigender Reihenfolge aus.
- Geben Sie die N Zahlen in einer zufälligen Reihenfolge aus.

1.1 Experimente

35. *Würfelsimulation*: Der folgende Code berechnet die genaue Wahrscheinlichkeitsverteilung für die Summe zweier Würfel:

```
int SIDES = 6;
double[] dist = new double[2*SIDES+1];
for (int i = 1; i <= SIDES; i++)
   for (int j = 1; j <= SIDES; j++)
      dist[i+j] += 1.0;

for (int k = 1; k <= 2*SIDES; k++)
   dist[k] /= 36.0;
```

Der Wert `dist[i]` ist die Wahrscheinlichkeit dafür, dass sich die Augenzahlen der Würfel zu `k` aufsummieren. Versuchen Sie, die berechneten Wahrscheinlichkeiten durch die Simulation eines *N*-maligen Würfelns zu bestätigen, und zählen Sie beim Aufsummieren der beiden zufälligen Integer zwischen 1 und 6 mit, wie oft jede Summe auftaucht. Wie groß muss *N* sein, bis Ihre empirischen Ergebnisse mit den exakten Ergebnissen bis auf drei Dezimalstellen übereinstimmen?

36. *Empirischer Test zur Qualitätsbestimmung einer Mischoperation*: Führen Sie Rechenexperimente durch, um zu prüfen, ob unser Mischcode aus Tabelle 1.11 auch wirklich wie angepriesen funktioniert. Schreiben Sie ein Programm `ShuffleTest`, das die Befehlszeilenargumente *M* und *N* entgegennimmt und *N*-mal ein Array der Größe *M* mischt (wobei das Array vor jedem Mischen mit `a[i] = i` initialisiert wird). Am Ende soll eine *M*×*M*-Tabelle ausgegeben werden, in der die Zeile `i` für alle `j` angibt, wie oft `i` nach dem Mischen auf Position `j` auftauchte. Alle Einträge im Array sollte nahe *N*/*M* sein.

37. *Schlechtes Mischen*: Was wäre, wenn Sie in unserem Mischcode einen zufälligen Integer auswählen würden, der zwischen 0 und `N-1` liegt anstatt zwischen `i` und `N-1`. Zeigen Sie, dass die *N*! möglichen Reihenfolgen *nicht* gleich wahrscheinlich sind. Führen Sie den Test der vorhergehenden Übung auch für diese Version aus.

38. *Binäre Suche versus Brute-Force-Suche*: Schreiben Sie ein Programm `BruteForceSearch`, das die Brute-Force-Suchmethode aus Abschnitt *Performance* verwendet, und vergleichen Sie seine Ausführungszeit auf Ihrem Computer mit der von `BinarySearch` unter Verwendung von *largeW.txt* und *largeT.txt*.

39. *Zufallsübereinstimmungen*: Schreiben Sie einen `BinarySearch`-Client, der einen `int`-Wert `T` als Befehlszeilenargument entgegennimmt und *T* Versuche des folgenden Experiments für $N=10^3$, 10^4, 10^5 und 10^6 durchführt: Erzeugen Sie zwei Arrays von *N* positiven sechsstelligen `int`-Zufallswerten und finden Sie die Anzahl der Werte, die in beiden Arrays vorhanden sind. Geben Sie eine Tabelle aus, die den Mittelwert für diese Zahl über die *T* Versuche für jeden Wert von *N* angibt.

1.2 Datenabstraktion

Ein Datentyp ist eine Kombination aus einer Wertemenge und einem Satz von Operationen, die für diese Werte definiert sind. Bisher haben wir vornehmlich die *primitiven* Datentypen von Java ausführlich abgehandelt: So sind zum Beispiel die *Werte* des primitiven Datentyps `int` die ganzen Zahlen zwischen -2^{31} und $2^{31}-1$ und als *Operationen* auf `int`-Werten stehen die grundlegenden arithmetischen und vergleichenden Operationen zur Verfügung, einschließlich `+`, `*`, `%`, `<` und `>`. Prinzipiell könnten wir alle unsere Programme allein mit den acht integrierten primitiven Datentypen schreiben; aber es ist wesentlich bequemer, Programme auf einer höheren Abstraktionsebene zu schreiben. In diesem Abschnitt konzentrieren wir uns auf den Prozess der Datentypdefinition und -verwendung, der auch als *Datenabstraktion* bezeichnet wird (als Ergänzung zur *Funktionsabstraktion*, die die Grundlage von Abschnitt 1.1 ist).

Programmieren in Java basiert größtenteils darauf, mithilfe des Java-Schlüsselworts `class` eigene Datentypen, sogenannte *Referenztypen*, zu erzeugen. Dieser Programmierstil wird als *objektorientierte Programmierung* bezeichnet, da er um das Konzept eines *Objekts* kreist – eine Entität, die einen Datentypwert enthält. Während man mit den primitiven Datentypen von Java im Grunde auf Programme beschränkt ist, die auf Zahlen operieren, können wir mithilfe der Referenztypen Programme schreiben, die auf Strings, Bildern, Sounds, und Hunderten von weiteren Abstraktionen operieren, die von den Java-Standardbibliotheken oder der Website zu diesem Buch zur Verfügung gestellt werden. Noch wichtiger als die Bibliotheken vordefinierter Datentypen ist allerdings, dass der Umfang der in der Java-Programmierung möglichen Datentypen unbegrenzt ist, *da Sie Ihre eigenen Datentypen definieren können*, um jede beliebige Abstraktion zu implementieren.

Ein *abstrakter Datentyp* (ADT) ist ein Datentyp, dessen Repräsentation vor dem Client verborgen ist. Die Implementierung eines abstrakten Datentyps als Java-Klasse unterscheidet sich nicht wesentlich von der Implementierung einer Funktionsbibliothek als Satz statischer Methoden. Der Hauptunterschied besteht darin, dass wir Daten mit den Funktionsimplementierungen verbinden und die Repräsentation der Daten vor dem Client verbergen. Bei der *Verwendung* eines abstrakten Datentyps gilt unser Interesse vor allem den *Operationen*, die in der API angegeben sind, und nicht der Datenrepräsentation. Bei der *Implementierung* eines abstrakten Datentyps gilt unser Interesse den *Daten*, auf die wir dann die Operationen implementieren.

Abstrakte Datentypen sind wichtig, da sie die Kapselung im Programmentwurf unterstützen. Wir verwenden sie in diesem Buch als Mittel,

- um Probleme präzise in Form von APIs zu spezifizieren, die von verschiedenen Clients genutzt werden können.
- um Algorithmen und Datenstrukturen als API-Implementierungen zu beschreiben.

Der Hauptgrund, warum wir verschiedene Algorithmen für die gleiche Aufgabe untersuchen, ist, dass sich deren Laufzeitverhalten unterscheidet. Abstrakte Datentypen eignen sich besonders gut für das Studium von Algorithmen, da sie uns erlauben, unsere

Erkenntnisse im Bereich der Algorithmusperformance direkt umzusetzen – einfach indem wir einen Algorithmus durch einen anderen ersetzen und so die Laufzeit für alle Clients verbessern, ohne dass irgendwelcher Client-Code geändert werden muss.

1.2.1 Abstrakte Datentypen

Ausgehend von der Regel, *dass man nicht wissen muss, wie ein Datentyp implementiert ist, um ihn nutzen zu können*, beschreiben wir zuerst, wie Sie Programme schreiben, die einen einfachen Datentyp namens `Counter` verwenden. Die Werte dieses Datentyps sind ein Name und eine nicht-negative ganze Zahl, seine Operationen bestehen darin, *den aktuellen Wert bei der Erzeugung mit null zu initialisieren, um eins zu inkrementieren und abzufragen*. Diese Abstraktion ist in vielen Bereichen nützlich. So könnten Sie einen solchen Datentyp zum Beispiel in elektronischer Wahlsoftware verwenden um sicherzustellen, dass ein Wähler nur das Wahlergebnis des von ihm gewählten Kandidaten um eins erhöhen kann. Oder Sie könnten einen `Counter` einsetzen, um grundlegende Operationen nachzuverfolgen, wenn Sie die Laufzeit von Algorithmen analysieren. Um `Counter` verwenden zu können, müssen Sie sich erstens mit der Art und Weise vertraut machen, wie wir die definierten Operationen angeben und beschreiben, zweitens müssen Sie den Java-Mechanismus zur Erzeugung und Manipulation von Datentypwerten kennenlernen. Beide Techniken sind wichtige Säulen einer modernen Programmierung und werden im ganzen Buch hindurch verwendet. Sie sollten dieses erste Beispiel daher besonders sorgfältig lesen.

API für einen abstrakten Datentyp

Um das Verhalten eines abstrakten Datentyps zu beschreiben, verwenden wir eine Programmierschnittstelle oder API (*Applications Programming Interface*), die aus einer Liste von *Konstruktoren* und *Instanzmethoden* (Operationen) besteht. Zusätzlich geben wir zu jedem Element der API eine formlose Beschreibung an, was für die API zu `Counter` wie folgt aussieht:

Tabelle 1.22 Eine API für einen Zähler

public class Counter

	`Counter(String id)`	Erzeugt einen Zähler namens `id`.
`void`	`increment()`	Inkrementiert den Zähler um eins.
`int`	`tally()`	Anzahl der Inkrementierungen seit der Erzeugung.
`String`	`toString()`	String-Repräsentation

Auch wenn die Grundlage einer Datentypdefinition eine Wertemenge ist, können Sie der API nicht die Rolle der Werte entnehmen, sondern nur die Operationen auf diese Werte. Insofern hat die Definition eines abstrakten Datentyps starke Ähnlichkeit mit einer Bibliothek statischer Methoden (siehe Abschnitt *Eigenschaften von Methoden*):

- Beide werden als Java-Klasse mit dem Schlüsselwort `class` implementiert.
- Instanzmethoden können null oder mehr Argumente eines angegebenen Typs entgegennehmen, die durch Kommata voneinander getrennt und in Klammern gesetzt sind.
- Sie können einen Rückgabewert eines angegebenen Typs vorsehen oder mit `void` signalisieren, dass es keinen Rückgabewert gibt.

Außerdem gibt es drei wichtige Unterschiede:

- Einige Einträge in der API weisen den gleichen Namen wie die Klasse auf und haben keinen Rückgabetyp. Diese Einträge werden als *Konstruktor* bezeichnet und spielen eine besondere Rolle. In unserem Fall hat `Counter` einen Konstruktor, der ein `String`-Argument übernimmt.
- Den Instanzmethoden fehlt der Modifizierer `static`, was darauf zurückzuführen ist, dass es *keine* statischen Methoden sind – ihre Aufgabe ist es, auf den Datentypwerten zu operieren.
- Einige Instanzmethoden sind entsprechend den Java-Konventionen bereits vorhanden – wir bezeichnen solche Methoden als *geerbte Methoden*.

Wie die APIs für Bibliotheken von statischen Methoden ist eine API für einen abstrakten Datentyp ein Vertrag mit allen Clients und deshalb der Ausgangspunkt sowohl für die Entwicklung von Client-Code als auch die Entwicklung von Datentypimplementierungen. In diesem Fall teilt uns die API mit, dass jedem Benutzer von `Counter` der Konstruktor `Counter()`, die Instanzmethoden `increment()` und `tally()` sowie die geerbte Methode `toString()` zur Verfügung stehen.

Geerbte Methoden

Diverse Java-Konventionen ermöglichen einem Datentyp, von den in Java integrierten Sprachmechanismen Gebrauch zu machen – einfach indem bestimmte Methoden in die API übernommen werden. So *erben* beispielsweise alle Datentypen in Java eine Methode `toString()`, die eine String-Repräsentation der Datentypwerte zurückliefert. Java ruft diese Methode auf, wenn einer dieser Datentypwerte mittels eines +-Operators mit einem `String`-Wert konkateniert, d.h. verkettet werden soll. Die Standardimplementierung ist nicht besonders nützlich (sie liefert eine String-Repräsentation der Speicheradresse des Datentypwerts), sodass wir in der Regel eine Implementierung anbieten, die den Standard überschreibt. Diese Implementierung nehmen wir dann natürlich in die API auf. Zu diesen Methoden gehören neben `toString()` auch `equals()`, `compareTo()` und `hashCode()` (▶Tabelle 1.40).

Client-Code

Wie bei der modularen Programmierung, die auf statischen Methoden basiert, können wir dank der APIs Client-Code schreiben, ohne Einzelheiten der Implementierung kennen zu müssen (und Implementierungscode schreiben, ohne Einzelheiten über einen bestimmten Client kennen zu müssen). Die in Tabelle 1.7 kurz zusammengefassten Mechanismen zum Verwalten von Programmen als unabhängige Module gelten für alle Java-Klassen und sind somit nicht nur für Bibliotheken statischer Methoden, sondern

auch für die modulare Programmierung mit abstrakten Datentypen nützlich und effektiv. Wir können also einen abstrakten Datentyp in jedem Programm verwenden, vorausgesetzt der Quellcode steht in einer *.java*-Datei im selben Verzeichnis oder in der Java-Standardbibliothek oder kann über eine `import`-Anweisung oder mittels einer der Klassenpfadmechanismen, die auf der Website zum Buch beschrieben sind, eingebunden werden. Anschließend kommen Sie in den Genuss aller Vorteile der modularen Programmierung. Indem wir den ganzen Code für die Implementierung eines Datentyps in einer Java-Klasse kapseln, ist es möglich, den Client-Code auf einer höheren Abstraktionsebene zu entwickeln. Für die Entwicklung von Client-Code müssen wir folgende Voraussetzungen erfüllen: Wir müssen *Variablen deklarieren*, *Objekte erzeugen*, die Datentypwerte enthalten, und den Instanzmethoden *Zugriff auf die Werte bieten*, damit sie darauf operieren können. Diese Prozesse unterscheiden sich von den vergleichbaren Prozessen für primitive Typen, auch wenn Ihnen viele Ähnlichkeiten auffallen werden.

Objekte

Eine Variable `heads` zu deklarieren, die mit Daten des Typs `Counter` verbunden werden kann, ist nicht schwer:

```
Counter heads;
```

Aber wie weisen Sie ihr Werte oder spezielle Operationen zu? Die Antwort auf diese Frage ist untrennbar mit einem Grundbegriff der Datenabstraktion verbunden: Ein *Objekt* ist eine Entität, die einen Datentypwert annehmen kann. Objekte lassen sich durch drei wesentliche Eigenschaften beschreiben: *Zustand*, *Identität* und *Verhalten*. Der *Zustand* eines Objekts ist ein Wert seines Datentyps. Die *Identität* eines Objekts unterscheidet es von allen anderen Objekten. Am besten stellen Sie sich darunter den Ort vor, an dem das Objekt im Speicher abgelegt ist. Das *Verhalten* eines Objekts wird durch die Operationen des Datentyps definiert. Es liegt allein in der Verantwortung der Implementierung, die Identität eines Objekts zu verwalten, sodass der Client-Code einen Datentyp verwenden kann, ohne Rücksicht auf die Repräsentation des Objektzustands nehmen zu müssen. Der Code muss sich lediglich an die API halten, die das Verhalten eines Objekts beschreibt. Ein Objekt kann einem Client Informationen über seinen Zustand liefern oder einen Nebeneffekt verursachen oder durch eine der Operationen seines Datentyps geändert werden, wobei jedoch die Details der Repräsentation des Datentypwerts für den Client-Code nicht wichtig sind. Für den Zugriff auf Objekte wird ein Verweismechanismus verwendet. Die Java-Nomenklatur bezeichnet deshalb nicht-primitive Datentypen auch als *Referenztypen*, um sie deutlich

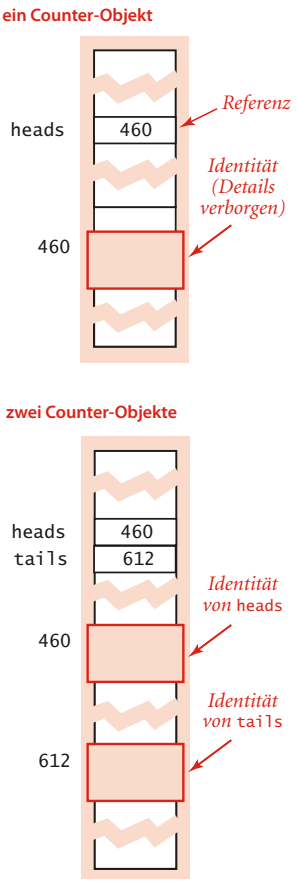


Abbildung 1.12: Objektrepräsentation

gegen die primitiven Datentypen abzugrenzen Wie genau die Verweise bzw. Referenzen implementiert sind, variiert von Java-Implementierung zu Java-Implementierung, aber es hilft, sich eine Referenz als eine Speicheradresse wie in ▶Abbildung 1.12 vorzustellen (aus Platzgründen verwenden wir im Diagramm dreistellige Speicheradressen).

Objekte erzeugen

Jeder Datentypwert ist in einem Objekt gespeichert. Um ein individuelles Objekt zu erzeugen (oder zu *instanziieren*), rufen wir einen Konstruktor mit dem Schlüsselwort `new` auf, gefolgt von dem Klassennamen und `()` (oder einer Liste von Argumentwerten in Klammern, wenn der Konstruktor Argumente entgegennimmt). Ein Konstruktor hat keinen Rückgabetyp, da er immer eine Referenz auf ein Objekt seines Datentyps zurückliefert. Jedes Mal, wenn ein Client `new()` verwendet, macht das System Folgendes:

- Es weist dem Objekt Speicher zu.
- Es ruft den Konstruktor auf, um seinen Wert zu initialisieren.
- Es liefert eine Referenz auf das Objekt zurück.

In Client-Code erzeugen wir Objekte normalerweise in einer initialisierenden Deklaration, die eine Variable mit dem Objekt verbindet – so wie wir es auch bei Variablen primitiver Datentypen oft machen. Im Gegensatz zu primitiven Datentypen werden Variablen mit Referenzen auf Objekte und nicht mit den Datentypwerten selbst verbunden. Wir können von derselben Klasse beliebig viele Objekte erzeugen; jedes Objekt hat seine eigene Identität und kann den gleichen (oder einen anderen) Wert wie ein anderes Objekt des gleichen Typs speichern. So erzeugt zum Beispiel der Code

```
Counter heads = new Counter("heads");
Counter tails = new Counter("tails");
```

zwei verschiedene `Counter`-Objekte. In einem abstrakten Datentyp bleiben die Details der Repräsentation des Wertes vor dem Client-Code verborgen. Sie können zwar vermuten, dass der Wert, der mit jedem `Counter`-Objekt verbunden ist, einen `String`-Namen und eine `int`-Zählung enthält, *aber Sie dürfen keinen Code schreiben, der davon ausgeht, dass die Werte eine bestimmte Repräsentation haben* (da Sie nicht sicher sein können, dass Ihre Annahme wahr ist – vielleicht ist die Zählung ja ein `long`-Wert).

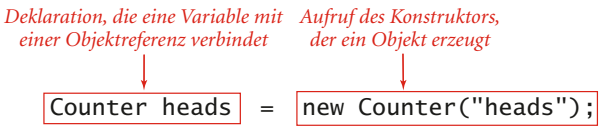


Abbildung 1.13: Ein Objekt erzeugen

Instanzmethoden aufrufen

Instanzmethoden dienen dem Zweck, auf Datentypwerten zu operieren. Aus diesem Grunde gibt es in Java zum Aufrufen von Instanzmethoden einen besonderen Mechanismus, der die Verbindung zu einem Objekt herstellt. Konkret sieht der Aufruf einer Instanzmethode so aus, dass wir einen Variablennamen schreiben, der auf ein Objekt verweist, gefolgt von einem Punkt, dem Namen einer Instanzmethode und eventuellen Argumenten, die in Klammern stehen und durch Kommata getrennt werden. Eine Instanzmethode kann den Datentypwert *ändern* oder ihn einfach *abrufen*. Instanzmethoden haben alle Eigenschaften von statischen Methoden, die wir in Abschnitt 1.4 beschrieben haben: Argumente werden als Wert übergeben, Methodennamen können überladen werden, die Methode kann einen Rückgabewert haben und sie kann Nebeneffekte verursachen. Darüber hinaus haben Instanzmethoden noch eine weitere Eigenschaft, die für sie spezifisch ist: *Jeder Aufruf ist mit einem Objekt verbunden*. So ruft beispielsweise der Code

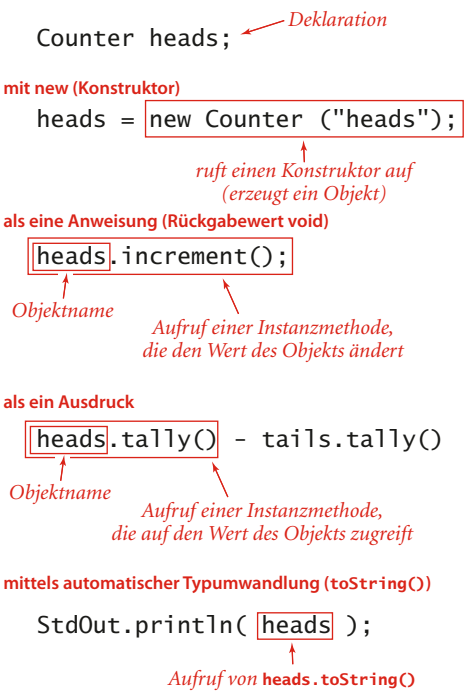


Abbildung 1.14: Instanzmethoden aufrufen

```
heads.increment();
```

die Instanzmethode `increment()` auf, die auf dem `Counter`-Objekt `heads` operiert (in diesem Fall erhöht die Operation die Zählung). Und der folgende Code

```
heads.tally() - tails.tally();
```

ruft zweimal die Instanzmethode `tally()` auf, die beim ersten Mal auf dem `Counter`-Objekt `heads` und beim zweiten Mal auf dem `Counter`-Objekt `tails` operiert (in diesem Fall liefert die Operation die Zählung als `int`-Wert zurück). Wie diese Beispiele zeigen, können Sie in Client-Code die Aufrufe von Instanzmethoden genauso verwenden wie die Aufrufe von statischen Methoden – als Anweisungen (`void`-Methoden) oder als Werte in Ausdrücken (Methoden, die einen Wert zurückliefern). Der primäre Zweck von statischen Methoden ist die Implementierung von Funktionen; der primäre Zweck von nicht-statischen (Instanz-)Methoden ist die Implementierung von Datentypoperationen. In Client-Code finden Sie beide Arten von Methoden, die Sie aber in unserem Client-Code leicht unterscheiden können, da der Aufruf einer statischen Methode immer mit einem *Klassennamen* (per Konvention am Anfang großgeschrieben) beginnt und der Aufruf einer nicht-statischen Methode immer mit einem *Objektnamen* (per Konvention am Anfang kleingeschrieben). Diese Unterschiede sind in ▶Tabelle 1.23 zusammengefasst.

Tabelle 1.23 Instanzmethoden im Vergleich zu statischen Methoden

	Instanzmethode	Statische Methode
Beispielaufruf	`head.increment()`	`Math.sqrt(2.0)`
Aufgerufen mit	Objektname	Klassenname
Parameter	Verweis auf Objekt und Argument(e)	Argument(e)
Hauptzweck	Untersucht oder ändert Objektwert	Berechnet den Rückgabewert

Objekte verwenden

Deklarationen liefern uns Variablennamen für Objekte. Diese Namen können wir im Code dazu nutzen, erzeugte Objekte zu referenzieren und Instanzmethoden aufzurufen; wir können sie aber auch ganz so wie Variablennamen für Integer, Gleitkommazahlen und andere primitive Datentypen verwenden. Wenn wir Client-Code für einen gegebenen Datentyp schreiben,

- deklarieren wir Variablen dieses Typs, über die wir Objekte referenzieren.
- rufen wir mit dem Schlüsselwort `new` einen Konstruktor auf, der Objekte des Typs erzeugt.
- rufen wir über den Variablennamen Instanzmethoden auf, und zwar entweder als Anweisungen oder als Teil von Ausdrücken.

So ist beispielsweise die Klasse `Flips` in ▶Listing 1.5 ein `Counter`-Client, der ein Befehlszeilenargument `T` übernimmt und `T` Münzwürfe simuliert (sie ist auch ein `StdRandom`-Client). Neben diesen direkten Einsatzbereichen können wir Variablen, die mit Objekten verbunden sind, genauso verwenden wie Variablen, die mit Werten eines primitiven Typs verbunden sind, und zwar:

- in Zuweisungen
- um Objekte an Methoden zu übergeben oder daraus zurückzuliefern
- um Arrays von Objekten zu erzeugen und zu verwenden

Um das Verhalten dieser jeweiligen Einsatzmöglichkeiten zu verstehen, müssen Sie in Form von *Referenzen* und nicht von Werten denken, wie Sie in der anschließenden Besprechung der obigen Punkte noch sehen werden.

Zuweisungen

Eine Zuweisung mit einem Referenztyp erzeugt eine Kopie der Referenz. Die Zuweisung erzeugt kein neues Objekt, sondern nur eine weitere Referenz auf ein bestehendes Objekt. Diese Situation wird als *Aliasing* bezeichnet: Beide Variablen beziehen sich auf dasselbe Objekt. Der Effekt des Aliasing entspricht vielleicht nicht Ihren Erwartungen, da er sich von dem Verhalten von Variablen unterscheidet, die Werte eines primitiven Typs enthalten. Bemühen Sie sich daher unbedingt, den Unterschied richtig zu verstehen.

```
public class Flips
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
              heads.increment();
         else tails.increment();
      StdOut.println(heads);
      StdOut.println(tails);
      int d = heads.tally() - tails.tally();
      StdOut.println("delta: " + Math.abs(d));
   }
}
```

Listing 1.5: `Counter`-Client zur Simulation von `T` Münzwürfen

```
% java Flips 10
5 heads
5 tails
delta: 0

% java Flips 10
8 heads
2 tails
delta: 6

% java Flips 1000000
499710 heads
500290 tails
delta: 580
```

Wenn `x` und `y` Variablen eines primitiven Typs sind, dann kopiert die Zuweisung `x = y` den Wert von `y` nach `x`. Bei Referenztypen wird die *Referenz* kopiert (nicht der Wert). Aliasing ist eine häufige Fehlerquelle in Java-Programmen, wie das folgende Beispiel demonstriert:

```
Counter c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();
StdOut.println(c1);
```

In Kombination mit einer typischen `toString()`-Implementierung würde dieser Code den String `"2 ones"` ausgeben, was möglicherweise nicht dem entspricht, was der Programmierer beabsichtigt hat, und zunächst kontraintuitiv ist. Solche Fehler kommen häufig in Programmen von Programmierern vor, die nur wenig Erfahrung in der Verwendung von Objekten haben. (Vielleicht zählen Sie ja auch dazu, also Achtung!) Änderungen am Zustand eines Objekts haben Auswirkungen auf allen Code mit Alias-

Variablen, die dieses Objekt referenzieren. Wir sind daran gewöhnt, zwei verschiedene Variablen von primitiven Typen als unabhängig voneinander zu betrachten, aber diese Anschauung lässt sich nicht auf Variablen von Referenztypen übertragen.

```
Counter c1;
c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();
```

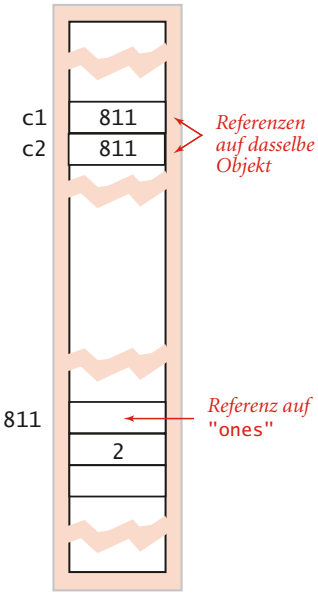


Abbildung 1.15: Aliasing

Objekte als Argumente

Sie können Objekte als *Argumente* an Methoden übergeben. Diese Möglichkeit vereinfacht normalerweise den Client-Code. Wenn wir zum Beispiel ein `Counter`-Objekt als Argument verwenden, übergeben wir im Prinzip einen Namen und eine Zahl, müssen aber nur eine Variable angeben. Wenn wir einer Methode beim Aufruf Argumente übergeben, ist dies in Java in etwa so, als stünde jeder der Argumentwerte auf der rechten Seite einer Zuweisung und der Name der entsprechenden Argumentvariablen auf der linken. Das heißt, Java übergibt eine *Kopie* des Argumentwertes vom aufrufenden Programm an die Methode. Dieser Mechanismus wird als *Wertübergabe* oder *Pass-by-Value* (Abschnitt *Eigenschaften von Methoden*) bezeichnet und hat zur Folge, dass die Methode den Wert von Variablen, die der Aufrufer als Argumente übergibt, nicht ändern kann. Für primitive Typen ist dies auch genau das, was wir erwarten würden (die beiden Variablen sind voneinander unabhängig), aber jedes Mal, wenn wir einen Referenztyp als Methodenargument verwenden, erzeugen wir ein Alias und müssen deshalb sehr vorsichtig sein. Mit anderen Worten, wir übergeben die *Referenz* als Wert (d.h. erstellen eine Kopie davon), aber das *Objekt* als Referenz. Wenn wir zum Beispiel eine Referenz auf ein Objekt vom Typ `Counter` übergeben, kann die Methode die ursprüngliche Referenz nicht ändern (sodass sie auf ein anderes `Counter`-Objekt zeigen würde), sie *kann* aber sehr wohl den Wert des Objekts ändern – beispielsweise indem sie über die Referenz `increment()` aufruft.

Objekte als Rückgabewerte

Selbstverständlich können Sie ein Objekt auch als *Rückgabewert* einer Methode verwenden. Die Methode könnte ein Objekt zurückliefern, das ihr wie im dazugehörigen Beispiel als Argument übergeben wurde, oder sie könnte ein Objekt erzeugen und eine Referenz darauf zurückliefern. Diese Fähigkeit ist wichtig, da Java-Methoden nur einen Rückgabewert zulassen – und durch die Verwendung von Objekten können wir Code schreiben, der faktisch mehrere Werte zurückliefert.

```
public class FlipsMax
{
   public static Counter max(Counter x, Counter y)
   {
      if (x.tally() > y.tally()) return x;
      else                       return y;
   }

   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
              heads.increment();
         else tails.increment();

      if (heads.tally() == tails.tally())
           StdOut.println("Tie");
      else StdOut.println(max(heads, tails) + " wins");
   }
}
```

Listing 1.6: Beispiel einer statischen Methode mit Objektargumenten und Rückgabewerten

```
% java FlipsMax 1000000
500281 tails wins
```

Arrays sind Objekte

In Java ist jeder Wert eines nicht-primitiven Typs ein Objekt. Dies gilt auch für Arrays, die in Java ebenfalls Objekte sind. Wie für Strings gibt es auch für bestimmte Operationen auf Arrays eine spezielle Sprachunterstützung: Dies betrifft Deklaration, Initialisierung und Indizierung. Wenn wir also einer Methode ein Array übergeben oder eine Arrayvariable auf der rechten Seite eine Zuweisung verwenden, machen wir das Gleiche wie bei jedem anderen Objekt, d.h., wir erstellen eine Kopie der Arrayreferenz und nicht eine Kopie des Arrays. Dies ist eine Konvention, die dem Umstand Rechnung trägt, dass Arrays häufig an Methoden übergeben werden, damit diese das Array durch Umstellung der Arrayelemente verändern – wie es zum Beispiel die Methoden `java.util.Arrays.sort()` oder `shuffle()` tun, die wir in Tabelle 1.11 betrachtet haben.

Arrays von Objekten

Arrayelemente können von jedem beliebigen Datentyp sein: `args[]` in unseren `main()`-Implementierungen ist ein Array von `String`-Objekten. Wenn wir ein Array von Objekten erstellen, geschieht dies in zwei Schritten:

- Wir erzeugen das Array, wobei wir uns der Klammer-Syntax für Arraykonstruktoren bedienen.
- Wir erzeugen die Objekte in dem Array, wozu wir normale Konstruktoren verwenden.

Der Code in ▶Listing 1.7 simuliert das Werfen eines Würfels, wobei in einem Array von `Counter`-Objekten die Anzahl der Vorkommen jedes möglichen Wertes gespeichert wird. Ein Array von Objekten ist in Java ein Array von Referenzen auf die Objekte, nicht der Objekte selbst. Für große Objekte verbessert dies die Effizienz, da wir nicht die Objekte, sondern ihre Referenzen verschieben müssen. Für kleine Objekte müssen wir einen gewissen Effizienzverlust in Kauf nehmen, da wir jedes Mal, wenn wir Informationen abrufen wollen, einer Referenz folgen müssen.

```
public class Rolls
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      int SIDES = 6;
      Counter[] rolls = new Counter[SIDES+1];
      for (int i = 1; i <= SIDES; i++)
         rolls[i] = new Counter(i + "'s");

      for (int t = 0; t < T; t++)
      {
         int result = StdRandom.uniform(1, SIDES+1);
         rolls[result].increment();
      }
      for (int i = 1; i <= SIDES; i++)
         StdOut.println(rolls[i]);
   }
}
```

Listing 1.7: `Counter`-Client, der `T` Würfe eines Würfels simuliert

```
% java Rolls 1000000
167308 1's
166540 2's
166087 3's
167051 4's
166422 5's
166592 6's
```

Bei diesem objektorientierten Ansatz wird das Schreiben von Code unter Verwendung von Datenabstraktion (Datentypen definieren und verwenden, mit Datentypwerten gespeichert in Objekten) im Allgemeinen als *objektorientierte Programmierung* bezeichnet. Die grundlegenden Konzepte, die wir gerade besprochen haben, bilden die Basis für die objektorientierte Programmierung, sodass wir sie hier kurz zusammenfassen wollen: Jeder *Datentyp* hat einen Wertebereich und einen dazugehörigen Satz an Operationen. Wir implementieren Datentypen in unabhängigen `class`-Modulen und schreiben Client-Programme, die diese dann verwenden. Ein *Objekt* ist demgegenüber eine Entität, die

einen Datentypwert oder eine *Instanz* eines Datentyps annehmen kann. Objekte lassen sich durch drei wesentliche Eigenschaften beschreiben: *Zustand*, *Verhalten* und *Identität*. Eine Datentypimplementierung unterstützt Clients des Datentyps wie folgt:

- Client-Code kann *Objekte erstellen* (d.h. Identitäten herstellen), indem er mit dem `new`-Konstrukt einen Konstruktor aufruft, der ein Objekt erzeugt, seine Instanzvariablen initialisiert und eine Referenz auf das Objekt zurückliefert.
- Client-Code kann *Datentypwerte manipulieren* (d.h. das Verhalten eines Objekts steuern, eventuell seinen Zustand ändern), indem er über eine Variable, die mit einem Objekt verbunden ist, eine Instanzmethode aufruft, die auf den Instanzvariablen des Objekts operiert.
- Client-Code kann *Objekte manipulieren*, indem er Arrays von Objekten erzeugt und diese an Methoden übergibt und zurückliefert, wie dies bei Werten primitiver Datentypen möglich ist. Allerdings mit dem Unterschied, dass sich die Variablen auf Referenzen zu Werten beziehen und nicht auf die Werte selbst.

Diese Mechanismen bilden das Fundament eines flexiblen, modernen und höchst nützlichen Programmierstils, auf den unser Studium der Algorithmen in diesem Buch fußt.

1.2.2 Beispiele abstrakter Datentypen

In Java gibt es Tausende von integrierten abstrakten Datentypen, die wir um viele weitere ergänzt haben, um Ihnen das Studium der Algorithmen zu erleichtern. Genau genommen ist jedes Java-Programm, das wir schreiben, eine Datentypimplementierung (oder eine Bibliothek von statischen Methoden). Um den Überblick zu behalten und Verwirrung vorzubeugen, werden wir für alle abstrakten Datentypen, die wir in diesem Buch verwenden, die APIs angeben. Zum Glück sind das nicht allzu viele.

In diesem Abschnitt werden wir Ihnen einige Datentypen, zusammen mit diversen Client-Code-Beispielen, exemplarisch vorstellen. (In einigen Fällen werden wir Ihnen die APIs, die Dutzende oder mehr Instanzmethoden enthalten können, nur auszugsweise vorstellen.) Auf diese Weise möchten wir Ihnen konkrete, aus der Praxis stammende Beispiele präsentieren, Ihnen die in diesem Buch verwendeten Instanzmethoden vorstellen und Ihnen noch einmal vor Augen führen, dass Sie die Details einer ADT-Implementierung nicht kennen müssen, um sie verwenden zu können.

Zum späteren Nachschlagen haben wir die Datentypen, die wir in diesem Buch verwenden und entwickeln, in ▶Tabelle 1.24 zusammengefasst. Dabei wird in mehrere Kategorien unterschieden:

- Standardsystem-ADTs in `java.lang.*`, die in jedem Java-Programm verwendet werden können.
- Java-ADTs in Bibliotheken wie `java.awt`, `java.net` und `java.io`, die ebenfalls in jedem Java-Programm verwendet werden können, aber eine `import`-Anweisung benötigen.
- Unsere E/A-ADTs, die mehrere Eingabe-/Ausgabeströme wie `StdIn` und `StdOut` unterstützen.

- Datenorientierte ADTs, deren Hauptzweck darin besteht, das Organisieren und Verarbeiten von Daten durch Kapselung der Repräsentation zu erleichtern. Wir beschreiben weiter hinten in diesem Abschnitt mehrere Anwendungen zu algorithmischer Geometrie und Datenverarbeitung und verwenden diese später als Beispiele in Client-Code.
- Collection-ADTs (Container-ADTs), deren Hauptzweck darin besteht, die Arbeit mit Sammlungen von Daten gleichen Typs zu erleichtern. Wir beschreiben die grundlegenden Typen `Bag`, `Stack` und `Queue` in Abschnitt 1.3, `PQ`-Typen in *Kapitel 2* und die Typen `ST` und `SET` in den *Kapiteln 3 und 5*.
- Operationsorientierte ADTs, die wir verwenden, um Algorithmen zu analysieren. Eine Beschreibung finden Sie in den Abschnitten 1.4 und 1.5.
- ADTs für Graph-Algorithmen, einschließlich den datenorientierten ADTs, die sich darauf konzentrieren, Repräsentationen von verschiedenen Arten von Graphen zu kapseln, sowie den operationsorientierten ADTs, die primär Spezifikationen für graphenverarbeitende Algorithmen bieten.

Diese Liste lässt jede Menge von Typen außer Acht, die wir in den Übungen betrachten (und die Sie zum Teil im Index nachschlagen können). Wie in Abschnitt 1.2.4 beschrieben, stellen wir oftmals mehrere Implementierungen eines abstrakten Datentyps vor, die wir dann durch entsprechende deskriptive Präfixe unterscheiden. Zusammengenommen beweisen die von uns verwendeten abstrakten Datentypen, dass das Organisieren und Verstehen der eingesetzten Datentypen ein ganz entscheidender Faktor in der modernen Programmierung ist.

Eine typische Anwendung verwendet unter Umständen nur fünf bis zehn dieser abstrakten Datentypen. Eines der Hauptziele bei der Entwicklung und Organisation von abstrakten Datentypen in diesem Buch ist es, Programmierer zu befähigen, beim Entwickeln von Client-Code mit relativ wenig von ihnen auszukommen.

Tabelle 1.24 Ausgewählte abstrakte Datentypen in diesem Buch

Standardsystem-Datentypen in java.lang	
`Integer`	`int`-Wrapper
`Double`	`double`-Wrapper
`String`	Indizierte Folge von `char`-Werten
`StringBuilder`	Erstellen von Strings
Weitere Java-Typen	
`java.awt.Color`	Farben
`java.awt.Font`	Schriftarten
`java.net.URL`	URLs
`java.io.File`	Dateien

Tabelle 1.24 Ausgewählte abstrakte Datentypen in diesem Buch *(Forts.)*

Unsere Standardtypen für die Ein-/Ausgabe	
`In`	Eingabestrom
`Out`	Ausgabestrom
`Draw`	Grafiken
Datenorientierte Typen für Client-Beispiele	
`Point2D`	Punkt in der Ebene
`Interval1D`	1-D-Intervall
`Interval2D`	2-D-Intervall
`Date`	Datum
`Transaction`	Transaktion
Typen für die Analyse von Algorithmen	
`Counter`	Zähler
`Accumulator`	Akkumulator
`VisualAccumulator`	Visuelle Version
`Stopwatch`	Stoppuhr
Collection-Typen	
`Stack`	Stapel
`Queue`	FIFO-Warteschlangen
`Bag`	Multimenge
`MinPQ  MaxPQ`	Vorrangwarteschlange
`IndexMinPQ  IndexMaxPQ`	Vorrangwarteschlange (indizierte)
`ST`	Symboltabelle
`SET`	Menge
`StringST`	Symboltabelle (Stringschlüssel)
Datenorientierte Typen für graphenverarbeitende Algorithmen	
`Graph`	Graph
`Digraph`	Gerichteter Graph
`Edge`	Kante (gewichtet)
`EdgeWeightedGraph`	Graph (gewichtet)
`DirectedEdge`	Kante (gerichtet, gewichtet)
`EdgeWeightedDigraph`	Graph (gerichtet, gewichtet)

Tabelle 1.24 Ausgewählte abstrakte Datentypen in diesem Buch *(Forts.)*

Operationsorientierte Typen für graphenverarbeitende Algorithmen	
`UF`	Verwaltung von Zusammenhangskomponenten
`DepthFirstPaths`	DFS-Pfadsuche
`CC`	Zusammenhangskomponenten
`BreadthFirstPaths`	BFS-Pfadsuche
`DirectedDFS`	DFS-Pfadsuche für gerichtete Graphen
`DirectedBFS`	BFS-Pfadsuche für gerichtete Graphen
`TransitiveClosure`	Alle Pfade
`Topological`	Topologische Reihenfolge
`DepthFirstOrder`	DFS-Reihenfolge
`DirectedCycle`	Zyklussuche
`SCC`	Starke Zusammenhangskomponenten
`MST`	Minimaler Spannbaum
`SP`	Kürzeste Pfade

Geometrische Objekte

Ein gutes Beispiel für objektorientierte Programmierung ist das Entwerfen von Datentypen für geometrische Objekte. Die APIs in ▶Tabelle 1.25 bis ▶Tabelle 1.27 definieren beispielsweise abstrakte Datentypen für drei vertraute geometrische Objekte: `Point2D` (Punkte in der Ebene), `Interval1D` (Intervalle auf der Linie) und `Interval2D` (zweidimensionale Intervalle in der Ebene oder achsenausgerichtete Rechtecke). Wie zuvor sind die APIs im Wesentlichen selbsterklärend und führen direkt zu leicht verständlichem Client-Code, wie das Beispiel in ▶Listing 1.8, das von der Befehlszeile die Grenzen eines `Interval2D`-Objekts und eine ganze Zahl T einliest, danach T zufällige Punkte im Einheitsquadrat erzeugt und die Anzahl der Punkte zählt, die im Intervall liegen (eine Schätzung der Rechteckfläche). Des dramatischen Effekts wegen liefert der Client auch noch eine visuelle Darstellung mit dem angegebenen Intervall und den Punkten, die außerhalb des Intervalls liegen. Diese Berechnung steht exemplarisch für eine Methode, die das Problem, Fläche und Volumen von geometrischen Figuren zu berechnen, auf das Problem reduziert festzustellen, ob ein Punkt innerhalb der Figur liegt oder nicht (ein weniger schwieriges, aber nicht triviales Problem). Wir könnten noch weitere APIs für andere geometrische Objekte wie Liniensegmente, Dreiecke, Polygone, Kreise usw. definieren, obwohl die Implementierung von Operationen auf diesen Objekten eine ziemliche Herausforderung sein kann (siehe Beispiele in den Übungen am Ende dieses Abschnitts).

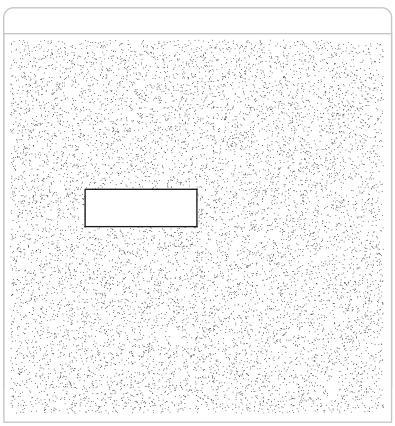
```
public static void main(String[] args)
{
   double xlo = Double.parseDouble(args[0]);
   double xhi = Double.parseDouble(args[1]);
   double ylo = Double.parseDouble(args[2]);
   double yhi = Double.parseDouble(args[3]);
   int T = Integer.parseInt(args[4]);

   Interval1D xinterval = new Interval1D(xlo, xhi);
   Interval1D yinterval = new Interval1D(ylo, yhi);
   Interval2D box = new Interval2D(xinterval, yinterval);
   box.draw();

   Counter c = new Counter("hits");
   for (int t = 0; t < T; t++)
   {
      double x = Math.random();
      double y = Math.random();
      Point2D p = new Point2D(x, y);
      if (box.contains(p)) c.increment();
      else                 p.draw();
   }

   StdOut.println(c);
   StdOut.println(box.area());
}
```

Listing 1.8: Interval2D-Testclient



```
% java Interval2D .2 .5 .5 .6 10000
297 hits
.03
```

Tabelle 1.25 Eine API für Punkte in der Ebene

public class Point2D

	`Point2D(double x, double y)`	Erzeugt einen Punkt
`double`	`x()`	*x*-Koordinate
`double`	`y()`	*y*-Koordinate
`double`	`r()`	Radius (Polarkoordinaten)
`double`	`theta()`	Winkel (Polarkoordinaten)
`double`	`distTo(Point2D that)`	Euklidische Distanz von diesem Punkt bis `that`
`void`	`draw()`	Zeichnet den Punkt mit `StdDraw`

Tabelle 1.26 Eine API für Intervalle auf der Linie

public class Interval1D

	`Interval1D(double lo, double hi)`	Erzeugt ein Intervall
`double`	`length()`	Länge des Intervalls
`boolean`	`contains(double x)`	Enthält das Intervall `x`?
`boolean`	`intersects(Interval1D that)`	Schneidet das Intervall `that`?

Tabelle 1.27 Eine API für zweidimensionale Intervalle in der Ebene

public class Interval2D

	`Interval2D(Interval1D x, Interval1D y)`	Erzeugt ein 2-D-Intervall
`double`	`area()`	Fläche des 2-D-Intervalls
`boolean`	`contains(Point2D p)`	Enthält das 2-D-Intervall `p`?
`boolean`	`intersects(Interval2D that)`	Schneidet das 2-D-Intervall `that`?

Programme, die geometrische Objekte verarbeiten, haben ein breites Anwendungsspektrum, das von Berechnungen mit Modellen der natürlichen Welt, wissenschaftlichem Rechnen bis zu Videospielen, Filmen und vielem mehr reicht. Die Entwicklung und das Studium solcher Programme und Anwendungen haben sich zu einem umfangreichen Studiengebiet entwickelt, das als *algorithmische Geometrie* bezeichnet wird und reich an Beispielen für die Anwendung der Algorithmen ist, die wir in diesem Buch betrachten. In diesem Kontext wollen wir Ihnen vor allem zeigen, dass abstrakte Datentypen, die direkt geometrische Abstraktionen repräsentieren, nicht schwer zu definieren sind und

zu einfachem und klarem Client-Code führen können. Diese Behauptung wollen wir durch zahlreiche Übungen am Ende dieses Abschnitts und auf der Website zum Buch untermauern.

Informationsverarbeitung

Egal, ob eine Bank Millionen von Kreditkartentransaktionen verarbeiten muss oder ein Webanalyse-Unternehmen Milliarden von Bildschirmklicks oder eine wissenschaftliche Forschungsgruppe Millionen von Testdaten – sehr viele Anwendungen dienen vor allem der Verarbeitung und Organisation von Daten. Und abstrakte Datentypen bieten einen natürlichen Mechanismus, diese Daten zu organisieren. Ohne allzu sehr ins Detail zu gehen, sei gesagt, dass die beiden APIs in ▶Tabelle 1.28 ein typischer Ansatz für kommerzielle Anwendungen sind. Der Grundgedanke ist, Datentypen zu definieren, die es uns erlauben, Daten in Objekten zu kapseln, die Dinge der realen Welt abbilden. Ein Datum besteht beispielsweise aus einem Tag, einem Monat und einem Jahr und eine Transaktion umfasst einen Kunden, ein Datum und einen Betrag. Dies sind nur zwei von vielen denkbaren Beispielen: Wir könnten genauso gut Datentypen definieren, die ausführliche Informationen über Kunden, Uhrzeiten, Orte, Waren, Dienstleistungen oder was auch immer halten. Jeder Datentyp besteht aus Konstruktoren, die Objekte mitsamt den Daten und Methoden erzeugen, auf die der Client-Code zugreifen kann. Um den Client-Code zu vereinfachen, sehen wir für jeden Typ zwei Konstruktoren vor – einen, der die für die Objekterzeugung nötigen Daten als einzelne Argumente entgegennimmt, und einen weiteren, der einen String parst, um die Daten einzulesen (Näheres hierzu finden Sie in *Übung 1.2.19*). Wie üblich muss der Client-Code nicht wissen, wie die Daten intern repräsentiert werden.

Der häufigste Grund, die Daten auf diese Weise zu organisieren, ist, dass die in einem Objekt zusammengefassten Daten als eine Einheit behandelt werden können: Wir können Arrays von `Transaction`-Werten verwalten, `Date`-Werte als Argument oder Rückgabewert für eine Methode verwenden usw. Beim Aufsetzen solcher Datentypen liegt der Fokus darauf, die Daten zu kapseln und gleichzeitig die Entwicklung von Client-Code zu erlauben, der nicht von der Repräsentation der Daten abhängt. Wir werden hier nicht näher auf diese Form der Datenorganisation eingehen, möchten aber noch darauf hinweisen, dass wir dank dieser Form der Datenorganisation und durch Hinzufügen der geerbten Methoden `toString()`, `compareTo()`, `equals()` und `hashCode()` Algorithmusimplementierungen nutzen können, die *jeden Datentyp* verarbeiten. Mit den geerbten Methoden werden wir uns noch etwas ausführlicher in Abschnitt *Schnittstellenvererbung* befassen. Die Java-Konvention, die es Clients erlaubt, von jedem Wert eine sinnvolle String-Repräsentation auszugeben, wenn wir in den Datentypen passende `toString()`-Implementierung aufnehmen, haben wir ja schon angesprochen. Mit den Konventionen zu den anderen geerbten Methoden werden wir uns in den Abschnitten 1.3, *2.5*, *3.4* und *3.5* auseinandersetzen und dabei `Transaction` und `Date` als Beispiele heranziehen. Abschnitt 1.3 behandelt klassische Datentypen und einen Java-Sprachmechanismus, der als *parametrisierte Typen* (oder *Generics*) bezeichnet wird und sich diese Konventionen zunutze macht. *Kapitel 2 und 3* nutzen ebenfalls die Vorteile generischer Typen und geerbter Methoden, mit dem Ziel, Implementierungen von Sortier- und Suchalgorithmen zu entwickeln, die für jeden Datentyp gelten.

Immer wenn Sie es mit Daten verschiedener Typen zu tun haben, die logisch zusammengehören, sollten Sie, wie in diesen Beispielen, die Definition eines abstrakten Datentyps in Erwägung ziehen. So können Sie die Daten besser organisieren, den Client-Code in den meisten Anwendungen stark vereinfachen und einen Riesenschritt in Richtung Datenabstraktion machen.

Tabelle 1.28 Beispiel-APIs für kommerzielle Anwendungen (Datumsangaben und Transaktionen)

public class Date implements Comparable<Date>		
	`Date(int month, int day, int year)`	Erzeugt ein Datum
	`Date(String date)`	Erzeugt ein Datum (Parser-Konstruktor)
`int`	`month()`	Monat
`int`	`day()`	Tag
`int`	`year()`	Jahr
`String`	`toString()`	String-Repräsentation
`boolean`	`equals(Object that)`	Ist dieses Datum das gleiche wie `that`?
`int`	`compareTo(Date that)`	Vergleicht dieses Datum mit `that`
`int`	`hashCode()`	Hashcode
public class Transaction implements Comparable<Transaction>		
	`Transaction(String who, Date when, double amount)`	Erzeugt eine Transaktion
	`Transaction(String transaction)`	Erzeugt eine Transaktion (Parser-Konstruktor)
`String`	`who()`	Kundenname
`Date`	`when()`	Datum
`double`	`amount()`	Betrag
`String`	`toString()`	String-Repräsentation
`boolean`	`equals(Object that)`	Ist diese Transaktion die gleiche wie `that`?
`int`	`compareTo(Transaction that)`	Vergleicht diese Transaktion mit `that`
`int`	`hashCode()`	Hashcode

Strings

Die Java-Klasse `String` ist ein besonders wichtiger und nützlicher abstrakter Datentyp, deren Objekte indizierte Folgen von `char`-Werten darstellen. Die Klasse `String` enthält jede Menge von Instanzmethoden, darunter: