

GLOBAL  
EDITION



# Building Java™ Programs

## *A Back to Basics Approach*

FOURTH EDITION

Stuart Reges • Marty Stepp



**Fourth Edition**  
**Global Edition**

# **Building Java Programs**

## **A Back to Basics Approach**

Stuart Reges  
*University of Washington*

Marty Stepp  
*Stanford University*



**Pearson**

---

Boston Columbus Indianapolis New York San Francisco Hoboken  
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto  
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

**Vice President, Editorial Director:** Marcia Horton  
**Acquisitions Editor:** Matt Goldstein  
**Editorial Assistant:** Kristy Alaura  
**Acquisitions Editor, Global Editions:** Sourabh Maheshwari  
**VP of Marketing:** Christy Lesko  
**Director of Field Marketing:** Tim Galligan  
**Product Marketing Manager:** Bram Van Kempen  
**Field Marketing Manager:** Demetrius Hall  
**Marketing Assistant:** Jon Bryant  
**Director of Product Management:** Erin Gregg  
**Team Lead, Program and Project Management:**  
Scott Disanno  
**Program Manager:** Carole Snyder  
**Project Editor, Global Editions:** K.K. Neelakantan

**Project Manager:** Lakeside Editorial Services L.L.C.  
**Senior Specialist, Program Planning and Support:**  
Maura Zaldivar-Garcia  
**Senior Manufacturing Controller, Global Editions:** Kay  
Holman  
**Media Production Manager, Global Editions:** Vikram  
Kumar  
**Cover Design:** Lumina Datamatics  
**R&P Manager:** Rachel Youdelman  
**R&P Project Manager:** Timothy Nicholls  
**Inventory Manager:** Meredith Maresca  
**Cover Art:** © Westend61 Premium/Shutterstock.com  
**Full-Service Project Management:**  
Apoorva Goel/Cenveo® Publisher Services

---

The authors and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in this book. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Acknowledgements of third-party content appear on pages 1219–1220, which constitute an extension of this copyright page.

PEARSON, and MYPROGRAMMINGLAB are exclusive trademarks in the U.S. and/or other countries owned by Pearson Education, Inc. or its affiliates.

Pearson Education Limited  
Edinburgh Gate  
Harlow  
Essex CM20 2JE  
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:  
[www.pearsonglobaleditions.com](http://www.pearsonglobaleditions.com)

© Pearson Education Limited 2018

The rights of Stuart Reges and Marty Stepp to be identified as the authors of this work have been asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

*Authorized adaptation from the United States edition, entitled Building Java Programs: A Back to Basics Approach, 4th Edition, ISBN 978-0-13-432276-6, by Stuart Reges and Marty Stepp published by Pearson Education © 2017.*

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a license permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

British Library Cataloguing-in-Publication Data  
A catalogue record for this book is available from the British Library  
10 9 8 7 6 5 4 3 2 1

ISBN 10: 1-292-16168-X  
ISBN 13: 978-1-292-16168-6

Typeset in Monotype by Cenveo Publisher Services  
Printed and bound in Malaysia.



The newly revised fourth edition of our *Building Java Programs* textbook is designed for use in a two-course introduction to computer science. We have class-tested it with thousands of undergraduates, most of whom were not computer science majors, in our CS1-CS2 sequence at the University of Washington. These courses are experiencing record enrollments, and other schools that have adopted our textbook report that students are succeeding with our approach.

Introductory computer science courses are often seen as “killer” courses with high failure rates. But as Douglas Adams says in *The Hitchhiker’s Guide to the Galaxy*, “Don’t panic.” Students can master this material if they can learn it gradually. Our textbook uses a layered approach to introduce new syntax and concepts over multiple chapters.

Our textbook uses an “objects later” approach where programming fundamentals and procedural decomposition are taught before diving into object-oriented programming. We have championed this approach, which we sometimes call “back to basics,” and have seen through years of experience that a broad range of scientists, engineers, and others can learn how to program in a procedural manner. Once we have built a solid foundation of procedural techniques, we turn to object-oriented programming. By the end of the course, students will have learned about both styles of programming.

Here are some of the changes that we have made in the fourth edition:

- **New chapter on functional programming with Java 8.** As explained below, we have introduced a chapter that uses the new language features available in Java 8 to discuss the core concepts of functional programming.
- **New section on images and 2D pixel array manipulation.** Image manipulation is becoming increasingly popular, so we have expanded our `DrawingPanel` class to include features that support manipulating images as two-dimensional arrays of pixel values. This extra coverage will be particularly helpful for students taking an AP/CS A course because of the heavy emphasis on two-dimensional arrays on the AP exam.
- **Expanded self-checks and programming exercises.** Many chapters have received new self-check problems and programming exercises. There are roughly fifty total problems and exercises per chapter, all of which have been class-tested with real students and have solutions provided for instructors on our web site.

Since the publication of our third edition, Java 8 has been released. This new version supports a style of programming known as functional programming that is gaining in

popularity because of its ability to simply express complex algorithms that are more easily executed in parallel on machines with multiple processors. ACM and IEEE have released new guidelines for undergraduate computer science curricula, including a strong recommendation to cover functional programming concepts.

We have added a new Chapter 19 that covers most of the functional concepts from the new curriculum guidelines. The focus is on concepts, not on language features. As a result, it provides an introduction to several new Java 8 constructs but not a comprehensive coverage of all new language features. This provides flexibility to instructors since functional programming features can be covered as an advanced independent topic, incorporated along the way, or skipped entirely. Instructors can choose to start covering functional constructs along with traditional constructs as early as Chapter 6. See the dependency chart at the end of this section.

The following features have been retained from previous editions:

- **Focus on problem solving.** Many textbooks focus on language details when they introduce new constructs. We focus instead on problem solving. What new problems can be solved with each construct? What pitfalls are novices likely to encounter along the way? What are the most common ways to use a new construct?
- **Emphasis on algorithmic thinking.** Our procedural approach allows us to emphasize algorithmic problem solving: breaking a large problem into smaller problems, using pseudocode to refine an algorithm, and grappling with the challenge of expressing a large program algorithmically.
- **Layered approach.** Programming in Java involves many concepts that are difficult to learn all at once. Teaching Java to a novice is like trying to build a house of cards. Each new card has to be placed carefully. If the process is rushed and you try to place too many cards at once, the entire structure collapses. We teach new concepts gradually, layer by layer, allowing students to expand their understanding at a manageable pace.
- **Case studies.** We end most chapters with a significant case study that shows students how to develop a complex program in stages and how to test it as it is being developed. This structure allows us to demonstrate each new programming construct in a rich context that can't be achieved with short code examples. Several of the case studies were expanded and improved in the second edition.
- **Utility as a CS1+CS2 textbook.** In recent editions, we added chapters that extend the coverage of the book to cover all of the topics from our second course in computer science, making the book usable for a two-course sequence. Chapters 12–19 explore recursion, searching and sorting, stacks and queues, collection implementation, linked lists, binary trees, hash tables, heaps, and more. Chapter 12 also

received a section on recursive backtracking, a powerful technique for exploring a set of possibilities for solving problems such as 8 Queens and Sudoku.

Layers and Dependencies

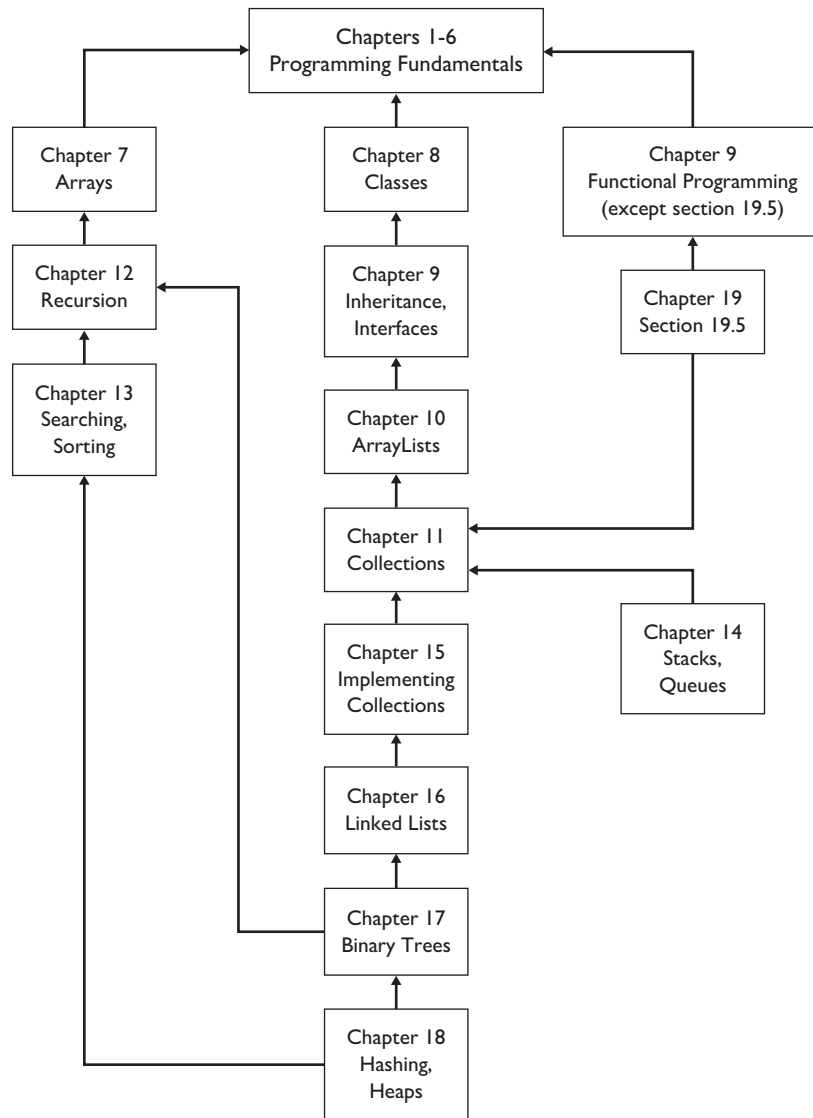
Many introductory computer science books are language-oriented, but the early chapters of our book are layered. For example, Java has many control structures (including for-loops, while-loops, and if/else-statements), and many books include all of these control structures in a single chapter. While that might make sense to someone who already knows how to program, it can be overwhelming for a novice who is learning how to program. We find that it is much more effective to spread these control structures into different chapters so that students learn one structure at a time rather than trying to learn them all at once.

The following table shows how the layered approach works in the first six chapters:

Chapter	Control Flow	Data	Programming Techniques	Input/Output
1	methods	String literals	procedural decomposition	println, print
2	definite loops (for)	variables, expressions, int, double	local variables, class constants, pseudocode	
3	return values	using objects	parameters	console input, 2D graphics (optional)
4	conditional (if/else)	char	pre/post conditions, throwing exceptions	printf
5	indefinite loops (while)	boolean	assertions, robust programs	
6		Scanner	token/line-based file processing	file I/O

Chapters 1–6 are designed to be worked through in order, with greater flexibility of study then beginning in Chapter 7. Chapter 6 may be skipped, although the case study in Chapter 7 involves reading from a file, a topic that is covered in Chapter 6.

The following is a dependency chart for the book:



## Supplements

Answers to all self-check problems appear on the web site and are accessible to anyone. Our web site has the following additional resources for students:

- **Online-only supplemental chapters**, such as a chapter on creating Graphical User Interfaces

- **Source code and data files** for all case studies and other complete program examples
- The **DrawingPanel** class used in the optional graphics Supplement 3G

Our web site has the following additional resources for teachers:

- **PowerPoint slides** suitable for lectures
- **Solutions** to exercises and programming projects, along with homework specification documents for many projects
- **Sample exams** and solution keys
- **Additional lab exercises** and **programming exercises** with solution keys
- **Closed lab creation tools** to produce lab handouts with the instructor's choice of problems integrated with the textbook

The materials are available at [www.pearsonglobaleditions.com/reges](http://www.pearsonglobaleditions.com/reges).

## MyProgrammingLab

MyProgrammingLab is an online practice and assessment tool that helps students fully grasp the logic, semantics, and syntax of programming. Through practice exercises and immediate, personalized feedback, MyProgrammingLab improves the programming competence of beginning students who often struggle with basic concepts and paradigms of popular high-level programming languages. A self-study and homework tool, the MyProgrammingLab course consists of hundreds of small practice exercises organized around the structure of this textbook. For students, the system automatically detects errors in the logic and syntax of code submissions and offers targeted hints that enable students to figure out what went wrong, and why. For instructors, a comprehensive grade book tracks correct and incorrect answers and stores the code inputted by students for review.

For a full demonstration, to see feedback from instructors and students, or to adopt MyProgrammingLab for your course, visit the following web site: <http://www.myprogramminglab.com/>

## VideoNotes



We have recorded a series of instructional videos to accompany the textbook. They are available at the following web site: [www.pearsonglobaleditions.com/reges](http://www.pearsonglobaleditions.com/reges).

Roughly 3–4 videos are posted for each chapter. An icon in the margin of the page indicates when a VideoNote is available for a given topic. In each video, we spend





- Eric Matson, Wright State University
- Kathryn S. McKinley, University of Texas, Austin
- Jerry Mead, Bucknell University
- George Medelinskas, Northern Essex Community College
- John Neitzke, Truman State University
- Dale E. Parson, Kutztown University
- Richard E. Pattis, Carnegie Mellon University
- Frederick Pratter, Eastern Oregon University
- Roger Priebe, University of Texas, Austin
- Dehu Qi, Lamar University
- John Rager, Amherst College
- Amala V.S. Rajan, Middlesex University
- Craig Reinhart, California Lutheran University
- Mike Scott, University of Texas, Austin
- Alexa Sharp, Oberlin College
- Tom Stokke, University of North Dakota
- Leigh Ann Sudol, Fox Lane High School
- Ronald F. Taylor, Wright State University
- Andy Ray Terrel, University of Chicago
- Scott Thede, DePauw University
- Megan Thomas, California State University, Stanislaus
- Dwight Tuinstra, SUNY Potsdam
- Jeannie Turner, Sayre School
- Tammy VanDeGrift, University of Portland
- Thomas John VanDrunen, Wheaton College
- Neal R. Wagner, University of Texas, San Antonio
- Jiangping Wang, Webster University
- Yang Wang, Missouri State University
- Stephen Weiss, University of North Carolina at Chapel Hill
- Laurie Werner, Miami University
- Dianna Xu, Bryn Mawr College
- Carol Zander, University of Washington, Bothell

Finally, we would like to thank the great staff at Pearson who helped produce the book. Michelle Brown, Jeff Holcomb, Maurene Goo, Patty Mahtani, Nancy Kotary, and Kathleen Kenny did great work preparing the first edition. Our copy editors and the staff of Aptara Corp, including Heather Sisan, Brian Baker, Brendan Short,

and Rachel Head, caught many errors and improved the quality of the writing. Marilyn Lloyd and Chelsea Bell served well as project manager and editorial assistant respectively on prior editions. For their help with the third edition we would like to thank Kayla Smith-Tarbox, Production Project Manager, and Jenah Blitz-Stoehr, Computer Science Editorial Assistant. Mohinder Singh and the staff at Aptara, Inc., were also very helpful in the final production of the third edition. For their great work on production of the fourth edition, we thank Louise Capulli and the staff of Lakeside Editorial Services, along with Carole Snyder at Pearson. Special thanks go to our lead editor at Pearson, Matt Goldstein, who has believed in the concept of our book from day one. We couldn't have finished this job without all of their hard work and support.

Stuart Reges  
Marty Stepp

## Acknowledgments for the Global Edition

Pearson would like to thank and acknowledge the following people for their contributions to the Global Edition.

### **Contributor**

Ankur Saxena, Amity University

### **Reviewers**

Arup Bhattacharya, RCC Institute of Technology

Soumen Mukherjee, RCC Institute of Technology

Khyat Sharma

# MyProgrammingLab™

Through the power of practice and immediate personalized feedback, MyProgrammingLab helps improve your students' performance.

## PROGRAMMING PRACTICE

With MyProgrammingLab, your students will gain first-hand programming experience in an interactive online environment.

## IMMEDIATE, PERSONALIZED FEEDBACK

MyProgrammingLab automatically detects errors in the logic and syntax of their code submission and offers targeted hints that enables students to figure out what went wrong and why.

## GRADUATED COMPLEXITY

MyProgrammingLab breaks down programming concepts into short, understandable sequences of exercises. Within each sequence the level and sophistication of the exercises increase gradually but steadily.

## DYNAMIC ROSTER

Students' submissions are stored in a roster that indicates whether the submission is correct, how many attempts were made, and the actual code submissions from each attempt.

## PEARSON eTEXT

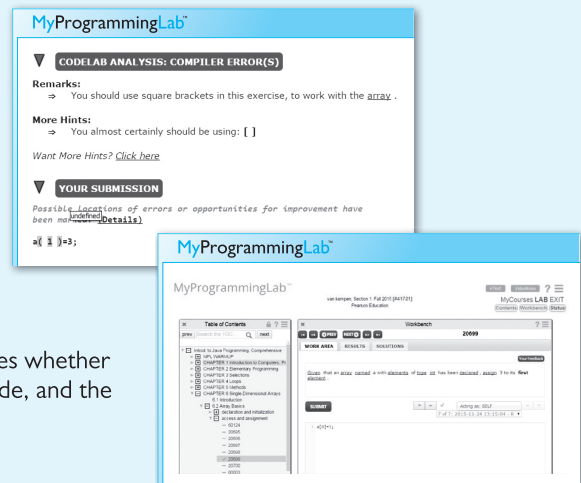
The Pearson eText gives students access to their textbook anytime, anywhere

## STEP-BY-STEP VIDEONOTE TUTORIALS

These step-by-step video tutorials enhance the programming concepts presented in select Pearson textbooks.

For more information and titles available with **MyProgrammingLab**, please visit [www.myprogramminglab.com](http://www.myprogramminglab.com).

Copyright © 2016 Pearson Education, Inc. or its affiliate(s). All rights reserved. HELO88173 • 11/15



**LOCATION OF VIDEO NOTES IN THE TEXT**[www.pearsonglobaleditions.com/reges](http://www.pearsonglobaleditions.com/reges)

VideoNote

<b>Chapter 1</b>	Pages 57, 66
<b>Chapter 2</b>	Pages 91, 100, 115, 123, 136
<b>Chapter 3</b>	Pages 167, 182, 187, 193
<b>Chapter 3G</b>	Pages 223, 241
<b>Chapter 4</b>	Pages 269, 277, 304
<b>Chapter 5</b>	Pages 350, 353, 355, 359, 382
<b>Chapter 6</b>	Pages 422, 435, 449
<b>Chapter 7</b>	Pages 484, 491, 510, 531
<b>Chapter 8</b>	Pages 561, 573, 581, 594
<b>Chapter 9</b>	Pages 623, 636, 652
<b>Chapter 10</b>	Pages 698, 703, 712
<b>Chapter 11</b>	Pages 742, 755, 763
<b>Chapter 12</b>	Pages 790, 798, 835
<b>Chapter 13</b>	Pages 860, 863, 869
<b>Chapter 14</b>	Pages 915, 922
<b>Chapter 15</b>	Pages 956, 962, 966
<b>Chapter 16</b>	Pages 998, 1005, 1018
<b>Chapter 17</b>	Pages 1063, 1064, 1074
<b>Chapter 18</b>	Pages 1099, 1118



## Brief Contents

<b>Chapter 1</b>	Introduction to Java Programming	27
<b>Chapter 2</b>	Primitive Data and Definite Loops	89
<b>Chapter 3</b>	Introduction to Parameters and Objects	163
<b>Supplement 3G</b>	Graphics (Optional)	222
<b>Chapter 4</b>	Conditional Execution	264
<b>Chapter 5</b>	Program Logic and Indefinite Loops	341
<b>Chapter 6</b>	File Processing	413
<b>Chapter 7</b>	Arrays	469
<b>Chapter 8</b>	Classes	556
<b>Chapter 9</b>	Inheritance and Interfaces	613
<b>Chapter 10</b>	ArrayLists	688
<b>Chapter 11</b>	Java Collections Framework	741
<b>Chapter 12</b>	Recursion	780
<b>Chapter 13</b>	Searching and Sorting	858
<b>Chapter 14</b>	Stacks and Queues	910
<b>Chapter 15</b>	Implementing a Collection Class	948
<b>Chapter 16</b>	Linked Lists	991
<b>Chapter 17</b>	Binary Trees	1043
<b>Chapter 18</b>	Advanced Data Structures	1097
<b>Chapter 19</b>	Functional Programming with Java 8	1133
<b>Appendix A</b>	Java Summary	1175
<b>Appendix B</b>	The Java API Specification and Javadoc Comments	1190
<b>Appendix C</b>	Additional Java Syntax	1196

This page intentionally left blank

# Contents

<b>Chapter 1 Introduction to Java Programming</b>	<b>27</b>
<b>1.1 Basic Computing Concepts</b>	<b>28</b>
Why Programming?	28
Hardware and Software	29
The Digital Realm	30
The Process of Programming	32
Why Java?	33
The Java Programming Environment	34
<b>1.2 And Now—Java</b>	<b>36</b>
String Literals (Strings)	40
<code>System.out.println</code>	41
Escape Sequences	41
<code>print</code> versus <code>println</code>	43
Identifiers and Keywords	44
A Complex Example: <code>DrawFigures1</code>	46
Comments and Readability	47
<b>1.3 Program Errors</b>	<b>50</b>
Syntax Errors	50
Logic Errors (Bugs)	54
<b>1.4 Procedural Decomposition</b>	<b>54</b>
Static Methods	57
Flow of Control	60
Methods That Call Other Methods	62
An Example Runtime Error	65
<b>1.5 Case Study: <code>DrawFigures</code></b>	<b>66</b>
Structured Version	67
Final Version without Redundancy	69
Analysis of Flow of Execution	70
<b>Chapter 2 Primitive Data and Definite Loops</b>	<b>89</b>
<b>2.1 Basic Data Concepts</b>	<b>90</b>
Primitive Types	90

Expressions	91
Literals	93
Arithmetic Operators	94
Precedence	96
Mixing Types and Casting	99
<b>2.2 Variables</b>	<b>100</b>
Assignment/Declaration Variations	105
String Concatenation	108
Increment/Decrement Operators	110
Variables and Mixing Types	113
<b>2.3 The for Loop</b>	<b>115</b>
Tracing for Loops	117
for Loop Patterns	121
Nested for Loops	123
<b>2.4 Managing Complexity</b>	<b>125</b>
Scope	125
Pseudocode	131
Class Constants	134
<b>2.5 Case Study: Hourglass Figure</b>	<b>136</b>
Problem Decomposition and Pseudocode	137
Initial Structured Version	139
Adding a Class Constant	140
Further Variations	143
<b>Chapter 3 Introduction to Parameters and Objects</b>	<b>163</b>
<b>3.1 Parameters</b>	<b>164</b>
The Mechanics of Parameters	167
Limitations of Parameters	171
Multiple Parameters	174
Parameters versus Constants	177
Overloading of Methods	177
<b>3.2 Methods That Return Values</b>	<b>178</b>
The Math Class	179
Defining Methods That Return Values	182
<b>3.3 Using Objects</b>	<b>186</b>
String Objects	187
Interactive Programs and Scanner Objects	193
Sample Interactive Program	196

<b>3.4 Case Study: Projectile Trajectory</b>	<b>199</b>
Unstructured Solution	203
Structured Solution	205

## **Supplement 3G Graphics (Optional) 222**

<b>3G.1 Introduction to Graphics</b>	<b>223</b>
DrawingPanel	223
Drawing Lines and Shapes	224
Colors	229
Drawing with Loops	232
Text and Fonts	236
Images	239
<b>3G.2 Procedural Decomposition with Graphics</b>	<b>241</b>
A Larger Example: DrawDiamonds	242
<b>3G.3 Case Study: Pyramids</b>	<b>245</b>
Unstructured Partial Solution	246
Generalizing the Drawing of Pyramids	248
Complete Structured Solution	249

## **Chapter 4 Conditional Execution 264**

<b>4.1 if/else Statements</b>	<b>265</b>
Relational Operators	267
Nested if/else Statements	269
Object Equality	276
Factoring if/else Statements	277
Testing Multiple Conditions	279
<b>4.2 Cumulative Algorithms</b>	<b>280</b>
Cumulative Sum	280
Min/Max Loops	282
Cumulative Sum with if	286
Roundoff Errors	288
<b>4.3 Text Processing</b>	<b>291</b>
The char Type	291
char versus int	292
Cumulative Text Algorithms	293
System.out.printf	295
<b>4.4 Methods with Conditional Execution</b>	<b>300</b>
Preconditions and Postconditions	300
Throwing Exceptions	300



Revisiting Return Values	304
Reasoning about Paths	309
<b>4.5 Case Study: Body Mass Index</b>	<b>311</b>
One-Person Unstructured Solution	312
Two-Person Unstructured Solution	315
Two-Person Structured Solution	317
Procedural Design Heuristics	321
 <b>Chapter 5 Program Logic and Indefinite Loops</b>	 <b>341</b>
<b>5.1 The while Loop</b>	<b>342</b>
A Loop to Find the Smallest Divisor	343
Random Numbers	346
Simulations	350
do/while Loop	351
<b>5.2 Fencepost Algorithms</b>	<b>353</b>
Sentinel Loops	355
Fencepost with if	356
<b>5.3 The boolean Type</b>	<b>359</b>
Logical Operators	361
Short-Circuited Evaluation	364
boolean Variables and Flags	368
Boolean Zen	370
Negating Boolean Expressions	373
<b>5.4 User Errors</b>	<b>374</b>
Scanner Lookahead	375
Handling User Errors	377
<b>5.5 Assertions and Program Logic</b>	<b>379</b>
Reasoning about Assertions	381
A Detailed Assertions Example	382
<b>5.6 Case Study: NumberGuess</b>	<b>387</b>
Initial Version without Hinting	387
Randomized Version with Hinting	389
Final Robust Version	393
 <b>Chapter 6 File Processing</b>	 <b>413</b>
<b>6.1 File-Reading Basics</b>	<b>414</b>
Data, Data Everywhere	414

Files and File Objects	414
Reading a File with a Scanner	417
<b>6.2 Details of Token-Based Processing</b>	<b>422</b>
Structure of Files and Consuming Input	424
Scanner Parameters	429
Paths and Directories	430
A More Complex Input File	433
<b>6.3 Line-Based Processing</b>	<b>435</b>
String Scanners and Line/Token Combinations	436
<b>6.4 Advanced File Processing</b>	<b>441</b>
Output Files with <code>PrintStream</code>	441
Guaranteeing That Files Can Be Read	446
<b>6.5 Case Study: Zip Code Lookup</b>	<b>449</b>
 <b>Chapter 7 Arrays</b>	 <b>469</b>
<b>7.1 Array Basics</b>	<b>470</b>
Constructing and Traversing an Array	470
Accessing an Array	474
A Complete Array Program	477
Random Access	481
Arrays and Methods	484
The For-Each Loop	487
Initializing Arrays	489
The <code>Arrays</code> Class	490
<b>7.2 Array-Traversal Algorithms</b>	<b>491</b>
Printing an Array	492
Searching and Replacing	494
Testing for Equality	497
Reversing an Array	498
String Traversal Algorithms	503
Functional Approach	504
<b>7.3 Reference Semantics</b>	<b>505</b>
Multiple Objects	507
<b>7.4 Advanced Array Techniques</b>	<b>510</b>
Shifting Values in an Array	510
Arrays of Objects	514
Command-Line Arguments	516
Nested Loop Algorithms	516

<b>7.5</b>	<b>Multidimensional Arrays</b>	<b>518</b>
	Rectangular Two-Dimensional Arrays	518
	Jagged Arrays	520
<b>7.6</b>	<b>Arrays of Pixels</b>	<b>525</b>
<b>7.7</b>	<b>Case Study: Benford's Law</b>	<b>530</b>
	Tallying Values	531
	Completing the Program	535
<b>Chapter 8</b>	<b>Classes</b>	<b>556</b>
<b>8.1</b>	<b>Object-Oriented Programming</b>	<b>557</b>
	Classes and Objects	558
	Point Objects	560
<b>8.2</b>	<b>Object State and Behavior</b>	<b>561</b>
	Object State: Fields	562
	Object Behavior: Methods	564
	The Implicit Parameter	567
	Mutators and Accessors	569
	The <code>toString</code> Method	571
<b>8.3</b>	<b>Object Initialization: Constructors</b>	<b>573</b>
	The Keyword <code>this</code>	578
	Multiple Constructors	580
<b>8.4</b>	<b>Encapsulation</b>	<b>581</b>
	Private Fields	582
	Class Invariants	588
	Changing Internal Implementations	592
<b>8.5</b>	<b>Case Study: Designing a Stock Class</b>	<b>594</b>
	Object-Oriented Design Heuristics	595
	<code>Stock</code> Fields and Method Headers	597
	<code>Stock</code> Method and Constructor Implementation	599
<b>Chapter 9</b>	<b>Inheritance and Interfaces</b>	<b>613</b>
<b>9.1</b>	<b>Inheritance Basics</b>	<b>614</b>
	Nonprogramming Hierarchies	615
	Extending a Class	617
	Overriding Methods	621

<b>9.2</b>	<b>Interacting with the Superclass</b>	<b>623</b>
	Calling Overridden Methods	623
	Accessing Inherited Fields	624
	Calling a Superclass's Constructor	626
	DividendStock Behavior	628
	The Object Class	630
	The equals Method	631
	The instanceof Keyword	634
<b>9.3</b>	<b>Polymorphism</b>	<b>636</b>
	Polymorphism Mechanics	639
	Interpreting Inheritance Code	641
	Interpreting Complex Calls	643
<b>9.4</b>	<b>Inheritance and Design</b>	<b>646</b>
	A Misuse of Inheritance	646
	Is-a Versus Has-a Relationships	649
	Graphics2D	650
<b>9.5</b>	<b>Interfaces</b>	<b>652</b>
	An Interface for Shapes	653
	Implementing an Interface	655
	Benefits of Interfaces	658
<b>9.6</b>	<b>Case Study: Financial Class Hierarchy</b>	<b>660</b>
	Designing the Classes	661
	Redundant Implementation	665
	Abstract Classes	668
<b>Chapter 10</b>	<b>ArrayLists</b>	<b>688</b>
<b>10.1</b>	<b>ArrayLists</b>	<b>689</b>
	Basic ArrayList Operations	690
	ArrayList Searching Methods	693
	A Complete ArrayList Program	696
	Adding to and Removing from an ArrayList	698
	Using the For-Each Loop with ArrayLists	702
	Wrapper Classes	703
<b>10.2</b>	<b>The Comparable Interface</b>	<b>706</b>
	Natural Ordering and compareTo	708
	Implementing the Comparable Interface	712
<b>10.3</b>	<b>Case Study: Vocabulary Comparison</b>	<b>718</b>
	Some Efficiency Considerations	718
	Version 1: Compute Vocabulary	721

Version 2: Compute Overlap	724
Version 3: Complete Program	729

## **Chapter 11 Java Collections Framework 741**

<b>11.1 Lists 742</b>	
Collections	742
LinkedList versus ArrayList	743
Iterators	746
Abstract Data Types (ADTs)	750
LinkedList Case Study: Sieve	752
<b>11.2 Sets 755</b>	
Set Concepts	756
TreeSet versus HashSet	758
Set Operations	759
Set Case Study: Lottery	761
<b>11.3 Maps 763</b>	
Basic Map Operations	764
Map Views (keySet and values)	766
TreeMap versus HashMap	767
Map Case Study: WordCount	768
Collection Overview	771

## **Chapter 12 Recursion 780**

<b>12.1 Thinking Recursively 781</b>	
A Nonprogramming Example	781
An Iterative Solution Converted to Recursion	784
Structure of Recursive Solutions	786
<b>12.2 A Better Example of Recursion 788</b>	
Mechanics of Recursion	790
<b>12.3 Recursive Functions and Data 798</b>	
Integer Exponentiation	798
Greatest Common Divisor	801
Directory Crawler	807
Helper Methods	811
<b>12.4 Recursive Graphics 814</b>	



<b>12.5 Recursive Backtracking</b>	<b>818</b>
A Simple Example: Traveling North/East	819
8 Queens Puzzle	824
Solving Sudoku Puzzles	831
<b>12.6 Case Study: Prefix Evaluator</b>	<b>835</b>
Infix, Prefix, and Postfix Notation	835
Evaluating Prefix Expressions	836
Complete Program	839

## **Chapter 13 Searching and Sorting** **858**

<b>13.1 Searching and Sorting in the Java Class Libraries</b>	<b>859</b>
Binary Search	860
Sorting	863
Shuffling	864
Custom Ordering with Comparators	865
<b>13.2 Program Complexity</b>	<b>869</b>
Empirical Analysis	870
Complexity Classes	876
<b>13.3 Implementing Searching and Sorting Algorithms</b>	<b>878</b>
Sequential Search	879
Binary Search	880
Recursive Binary Search	883
Searching Objects	886
Selection Sort	877
<b>13.4 Case Study: Implementing Merge Sort</b>	<b>890</b>
Splitting and Merging Arrays	891
Recursive Merge Sort	894
Complete Program	897

## **Chapter 14 Stacks and Queues** **910**

<b>14.1 Stack/Queue Basics</b>	<b>911</b>
Stack Concepts	911
Queue Concepts	914
<b>14.2 Common Stack/Queue Operations</b>	<b>915</b>
Transferring Between Stacks and Queues	917
Sum of a Queue	918
Sum of a Stack	919

<b>14.3</b>	<b>Complex Stack/Queue Operations</b>	<b>922</b>
	Removing Values from a Queue	922
	Comparing Two Stacks for Similarity	924
<b>14.4</b>	<b>Case Study: Expression Evaluator</b>	<b>926</b>
	Splitting into Tokens	927
	The Evaluator	932
<b>Chapter 15</b>	<b>Implementing a Collection Class</b>	<b>948</b>
<b>15.1</b>	<b>Simple <code>ArrayIntList</code></b>	<b>949</b>
	Adding and Printing	949
	Thinking about Encapsulation	955
	Dealing with the Middle of the List	956
	Another Constructor and a Constant	961
	Preconditions and Postconditions	962
<b>15.2</b>	<b>A More Complete <code>ArrayIntList</code></b>	<b>966</b>
	Throwing Exceptions	966
	Convenience Methods	969
<b>15.3</b>	<b>Advanced Features</b>	<b>972</b>
	Resizing When Necessary	972
	Adding an Iterator	974
<b>15.4</b>	<b><code>ArrayList&lt;E&gt;</code></b>	<b>980</b>
<b>Chapter 16</b>	<b>Linked Lists</b>	<b>991</b>
<b>16.1</b>	<b>Working with Nodes</b>	<b>992</b>
	Constructing a List	993
	List Basics	995
	Manipulating Nodes	998
	Traversing a List	1001
<b>16.2</b>	<b>A Linked List Class</b>	<b>1005</b>
	Simple <code>LinkedIntList</code>	1005
	Appending add	1007
	The Middle of the List	1011
<b>16.3</b>	<b>A Complex List Operation</b>	<b>1018</b>
	Inchworm Approach	1023
<b>16.4</b>	<b>An <code>IntList</code> Interface</b>	<b>1024</b>

<b>16.5</b>	<b>LinkedList&lt;E&gt;</b>	<b>1027</b>
	Linked List Variations	1028
	Linked List Iterators	1031
	Other Code Details	1033

## **Chapter 17 Binary Trees** **1043**

<b>17.1</b>	<b>Binary Tree Basics</b>	<b>1044</b>
	Node and Tree Classes	1047
<b>17.2</b>	<b>Tree Traversals</b>	<b>1048</b>
	Constructing and Viewing a Tree	1054
<b>17.3</b>	<b>Common Tree Operations</b>	<b>1063</b>
	Sum of a Tree	1063
	Counting Levels	1064
	Counting Leaves	1066
<b>17.4</b>	<b>Binary Search Trees</b>	<b>1067</b>
	The Binary Search Tree Property	1068
	Building a Binary Search Tree	1070
	The Pattern <code>x = change(x)</code>	1074
	Searching the Tree	1077
	Binary Search Tree Complexity	1081
<b>17.5</b>	<b>SearchTree&lt;E&gt;</b>	<b>1082</b>

## **Chapter 18 Advanced Data Structures** **1097**

<b>18.1</b>	<b>Hashing</b>	<b>1098</b>
	Array Set Implementations	1098
	Hash Functions and Hash Tables	1099
	Collisions	1101
	Rehashing	1106
	Hashing Non-Integer Data	1109
	Hash Map Implementation	1112
<b>18.2</b>	<b>Priority Queues and Heaps</b>	<b>1113</b>
	Priority Queues	1113
	Introduction to Heaps	1115
	Removing from a Heap	1117
	Adding to a Heap	1118
	Array Heap Implementation	1120
	Heap Sort	1124

<b>Chapter 19 Functional Programming with Java 8</b>	<b>1133</b>
<b>19.1 Effect-Free Programming</b>	<b>1134</b>
<b>19.2 First-Class Functions</b>	<b>1137</b>
Lambda Expressions	1140
<b>19.3 Streams</b>	<b>1143</b>
Basic Idea	1143
Using Map	1145
Using Filter	1146
Using Reduce	1148
Optional Results	1149
<b>19.4 Function Closures</b>	<b>1150</b>
<b>19.5 Higher-Order Operations on Collections</b>	<b>1153</b>
Working with Arrays	1154
Working with Lists	1155
Working with Files	1159
<b>19.6 Case Study: Perfect Numbers</b>	<b>1160</b>
Computing Sums	1161
Incorporating Square Root	1164
Just Five and Leveraging Concurrency	1167
<b>Appendix A Java Summary</b>	<b>1175</b>
<b>Appendix B The Java API Specification and Javadoc Comments</b>	<b>1190</b>
<b>Appendix C Additional Java Syntax</b>	<b>1196</b>
<b>Index</b>	<b>1205</b>
<b>Credits</b>	<b>1219</b>

# Introduction to Java Programming

## Introduction

---

This chapter begins with a review of some basic terminology about computers and computer programming. Many of these concepts will come up in later chapters, so it will be useful to review them before we start delving into the details of how to program in Java.

We will begin our exploration of Java by looking at simple programs that produce output. This discussion will allow us to explore many elements that are common to all Java programs, while working with programs that are fairly simple in structure.

After we have reviewed the basic elements of Java programs, we will explore the technique of procedural decomposition by learning how to break up a Java program into several methods. Using this technique, we can break up complex tasks into smaller subtasks that are easier to manage and we can avoid redundancy in our program solutions.

### 1.1 Basic Computing Concepts

- Why Programming?
- Hardware and Software
- The Digital Realm
- The Process of Programming
- Why Java?
- The Java Programming Environment

### 1.2 And Now—Java

- String Literals (Strings)
- `System.out.println`
- Escape Sequences
- `print` versus `println`
- Identifiers and Keywords
- A Complex Example: `DrawFigures1`
- Comments and Readability

### 1.3 Program Errors

- Syntax Errors
- Logic Errors (Bugs)

### 1.4 Procedural Decomposition

- Static Methods
- Flow of Control
- Methods That Call Other Methods
- An Example Runtime Error

### 1.5 Case Study: `DrawFigures`

- Structured Version
- Final Version without Redundancy
- Analysis of Flow of Execution



## 1.1 Basic Computing Concepts

Computers are pervasive in our daily lives, and, thanks to the Internet, they give us access to nearly limitless information. Some of this information is essential news, like the headlines at [cnn.com](http://cnn.com). Computers let us share photos with our families and map directions to the nearest pizza place for dinner.

Lots of real-world problems are being solved by computers, some of which don't much resemble the one on your desk or lap. Computers allow us to sequence the human genome and search for DNA patterns within it. Computers in recently manufactured cars monitor each vehicle's status and motion. Digital music players such as Apple's iPod actually have computers inside their small casings. Even the Roomba vacuum-cleaning robot houses a computer with complex instructions about how to dodge furniture while cleaning your floors.

But what makes a computer a computer? Is a calculator a computer? Is a human being with a paper and pencil a computer? The next several sections attempt to address this question while introducing some basic terminology that will help prepare you to study programming.

### Why Programming?

At most universities, the first course in computer science is a programming course. Many computer scientists are bothered by this because it leaves people with the impression that computer science is programming. While it is true that many trained computer scientists spend time programming, there is a lot more to the discipline. So why do we study programming first?

A Stanford computer scientist named Don Knuth answers this question by saying that the common thread for most computer scientists is that we all in some way work with *algorithms*.

#### Algorithm

A step-by-step description of how to accomplish a task.

Knuth is an expert in algorithms, so he is naturally biased toward thinking of them as the center of computer science. Still, he claims that what is most important is not the algorithms themselves, but rather the thought process that computer scientists employ to develop them. According to Knuth,

It has often been said that a person does not really understand something until after teaching it to someone else. Actually a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm.<sup>1</sup>

<sup>1</sup>Knuth, Don. *Selected Papers on Computer Science*. Stanford, CA: Center for the Study of Language and Information, 1996.

Knuth is describing a thought process that is common to most of computer science, which he refers to as *algorithmic thinking*. We study programming not because it is the most important aspect of computer science, but because it is the best way to explain the approach that computer scientists take to solving problems.

The concept of algorithms is helpful in understanding what a computer is and what computer science is all about. The Merriam-Webster dictionary defines the word “computer” as “one that computes.” Using that definition, all sorts of devices qualify as computers, including calculators, GPS navigation systems, and children’s toys like the Furby. Prior to the invention of electronic computers, it was common to refer to humans as computers. The nineteenth-century mathematician Charles Peirce, for example, was originally hired to work for the U.S. government as an “Assistant Computer” because his job involved performing mathematical computations.

In a broad sense, then, the word “computer” can be applied to many devices. But when computer scientists refer to a computer, we are usually thinking of a universal computation device that can be programmed to execute any algorithm. Computer science, then, is the study of computational devices and the study of computation itself, including algorithms.

Algorithms are expressed as computer programs, and that is what this book is all about. But before we look at how to program, it will be useful to review some basic concepts about computers.

## Hardware and Software

A computer is a machine that manipulates data and executes lists of instructions known as *programs*.

### Program

A list of instructions to be carried out by a computer.

One key feature that differentiates a computer from a simpler machine like a calculator is its versatility. The same computer can perform many different tasks (playing games, computing income taxes, connecting to other computers around the world), depending on what program it is running at a given moment. A computer can run not only the programs that exist on it currently, but also new programs that haven’t even been written yet.

The physical components that make up a computer are collectively called *hardware*. One of the most important pieces of hardware is the central processing unit, or *CPU*. The CPU is the “brain” of the computer: It is what executes the instructions. Also important is the computer’s *memory* (often called random access memory, or *RAM*, because the computer can access any part of that memory at any time). The computer uses its memory to store programs that are being executed, along with their data. RAM is limited in size and does not retain its contents when the computer is turned off. Therefore, computers generally also use a *hard disk* as a larger permanent storage area.

Computer programs are collectively called *software*. The primary piece of software running on a computer is its operating system. An *operating system* provides an environment in which many programs may be run at the same time; it also provides a bridge between those programs, the hardware, and the *user* (the person using the computer). The programs that run inside the operating system are often called *applications*.

When the user selects a program for the operating system to run (e.g., by double-clicking the program's icon on the desktop), several things happen: The instructions for that program are loaded into the computer's memory from the hard disk, the operating system allocates memory for that program to use, and the instructions to run the program are fed from memory to the CPU and executed sequentially.

## The Digital Realm

In the last section, we saw that a computer is a general-purpose device that can be programmed. You will often hear people refer to modern computers as *digital* computers because of the way they operate.

### Digital

Based on numbers that increase in discrete increments, such as the integers 0, 1, 2, 3, etc.

Because computers are digital, everything that is stored on a computer is stored as a sequence of integers. This includes every program and every piece of data. An MP3 file, for example, is simply a long sequence of integers that stores audio information. Today we're used to digital music, digital pictures, and digital movies, but in the 1940s, when the first computers were built, the idea of storing complex data in integer form was fairly unusual.

Not only are computers digital, storing all information as integers, but they are also *binary*, which means they store integers as *binary numbers*.

### Binary Number

A number composed of just 0s and 1s, also known as a base-2 number.

Humans generally work with *decimal* or base-10 numbers, which match our physiology (10 fingers and 10 toes). However, when we were designing the first computers, we wanted systems that would be easy to create and very reliable. It turned out to be simpler to build these systems on top of binary phenomena (e.g., a circuit being open or closed) rather than having 10 different states that would have to be distinguished from one another (e.g., 10 different voltage levels).

From a mathematical point of view, you can store things just as easily using binary numbers as you can using base-10 numbers. But since it is easier to construct a physical device that uses binary numbers, that's what computers use.

This does mean, however, that people who aren't used to computers find their conventions unfamiliar. As a result, it is worth spending a little time reviewing how binary

numbers work. To count with binary numbers, as with base-10 numbers, you start with 0 and count up, but you run out of digits much faster. So, counting in binary, you say

0  
1

And already you've run out of digits. This is like reaching 9 when you count in base-10. After you run out of digits, you carry over to the next digit. So, the next two binary numbers are

10  
11

And again, you've run out of digits. This is like reaching 99 in base-10. Again, you carry over to the next digit to form the three-digit number 100. In binary, whenever you see a series of ones, such as 11111, you know you're just one away from the digits all flipping to 0s with a 1 added in front, the same way that, in base-10, when you see a number like 999999, you know that you are one away from all those digits turning to 0s with a 1 added in front.

Table 1.1 shows how to count up to the base-10 number 8 using binary.

**Table 1.1** Decimal vs. Binary

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

We can make several useful observations about binary numbers. Notice in the table that the binary numbers 1, 10, 100, and 1000 are all perfect powers of 2 ( $2^0$ ,  $2^1$ ,  $2^2$ ,  $2^3$ ). In the same way that in base-10 we talk about a ones digit, tens digit, hundreds digit, and so on, we can think in binary of a ones digit, twos digit, fours digit, eights digit, sixteens digit, and so on.

Computer scientists quickly found themselves needing to refer to the sizes of different binary quantities, so they invented the term *bit* to refer to a single binary digit and the term *byte* to refer to 8 bits. To talk about large amounts of memory, they invented the terms “kilobytes” (KB), “megabytes” (MB), “gigabytes” (GB), and so on. Many people think that these correspond to the metric system, where “kilo” means 1000, but that is only approximately true. We use the fact that  $2^{10}$  is approximately equal to 1000 (it actually equals 1024). Table 1.2 shows some common units of memory storage:

Table 1.2 Units of Memory Storage

Measurement	Power of 2	Actual Value	Example
kilobyte (KB)	2 <sup>10</sup>	1024	500-word paper (3 KB)
megabyte (MB)	2 <sup>20</sup>	1,048,576	typical book (1 MB) or song (5 MB)
gigabyte (GB)	2 <sup>30</sup>	1,073,741,824	typical movie (4.7 GB)
terabyte (TB)	2 <sup>40</sup>	1,099,511,627,776	20 million books in the Library of Congress (20 TB)
petabyte (PB)	2 <sup>50</sup>	1,125,899,906,842,624	10 billion photos on Facebook (1.5 PB)

The Process of Programming

The word *code* describes program fragments (“these four lines of code”) or the act of programming (“Let’s code this into Java”). Once a program has been written, you can *execute* it.

Program Execution

The act of carrying out the instructions contained in a program.

The process of execution is often called *running*. This term can also be used as a verb (“When my program runs it does something strange”) or as a noun (“The last run of my program produced these results”).

A computer program is stored internally as a series of binary numbers known as the *machine language* of the computer. In the early days, programmers entered numbers like these directly into the computer. Obviously, this is a tedious and confusing way to program a computer, and we have invented all sorts of mechanisms to simplify this process.

Modern programmers write in what are known as high-level programming languages, such as Java. Such programs cannot be run directly on a computer: They first have to be translated into a different form by a special program known as a *compiler*.

Compiler

A program that translates a computer program written in one language into an equivalent program in another language (often, but not always, translating from a high-level language into machine language).

A compiler that translates directly into machine language creates a program that can be executed directly on the computer, known as an *executable*. We refer to such compilers as *native compilers* because they compile code to the lowest possible level (the native machine language of the computer).

This approach works well when you know exactly what computer you want to use to run your program. But what if you want to execute a program on many different

computers? You'd need a compiler that generates different machine language output for each of them. The designers of Java decided to use a different approach. They cared a lot about their programs being able to run on many different computers, because they wanted to create a language that worked well for the Web.

Instead of compiling into machine language, Java programs compile into what are known as *Java bytecodes*. One set of bytecodes can execute on many different machines. These bytecodes represent an intermediate level: They aren't quite as high-level as Java or as low-level as machine language. In fact, they are the machine language of a theoretical computer known as the *Java Virtual Machine (JVM)*.

#### Java Virtual Machine

A theoretical computer whose machine language is the set of Java bytecodes.

A JVM isn't an actual machine, but it's similar to one. When we compile programs to this level, there isn't much work remaining to turn the Java bytecodes into actual machine instructions.

To actually execute a Java program, you need another program that will execute the Java bytecodes. Such programs are known generically as *Java runtimes*, and the standard environment distributed by Oracle Corporation is known as the *Java Runtime Environment (JRE)*.

#### Java Runtime

A program that executes compiled Java bytecodes.

Most people have Java runtimes on their computers, even if they don't know about them. For example, Apple's Mac OS X includes a Java runtime, and many Windows applications install a Java runtime.

## Why Java?

When Sun Microsystems released Java in 1995, it published a document called a "white paper" describing its new programming language. Perhaps the key sentence from that paper is the following:

Java: A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.<sup>2</sup>

This sentence covers many of the reasons why Java is a good introductory programming language. For starters, Java is reasonably simple for beginners to learn, and it embraces object-oriented programming, a style of writing programs that has been shown to be very successful for creating large and complex software systems.

<sup>2</sup><http://www.oracle.com/technetwork/java/langenv-140151.html>

Java also includes a large amount of prewritten software that programmers can utilize to enhance their programs. Such off-the-shelf software components are often called *libraries*. For example, if you wish to write a program that connects to a site on the Internet, Java contains a library to simplify the connection for you. Java contains libraries to draw graphical user interfaces (GUIs), retrieve data from databases, and perform complex mathematical computations, among many other things. These libraries collectively are called the *Java class libraries*.

### Java Class Libraries

The collection of preexisting Java code that provides solutions to common programming problems.

The richness of the Java class libraries has been an extremely important factor in the rise of Java as a popular language. The Java class libraries in version 1.7 include over 4000 entries.

Another reason to use Java is that it has a vibrant programmer community. Extensive online documentation and tutorials are available to help programmers learn new skills. Many of these documents are written by Oracle, including an extensive reference to the Java class libraries called the *API Specification* (API stands for Application Programming Interface).

Java is extremely platform independent; unlike programs written in many other languages, the same Java program can be executed on many different operating systems, such as Windows, Linux, and Mac OS X.

Java is used extensively for both research and business applications, which means that a large number of programming jobs exist in the marketplace today for skilled Java programmers. A sample Google search for the phrase “Java jobs” returned around 180,000,000 hits at the time of this writing.

## The Java Programming Environment

You must become familiar with your computer setup before you start programming. Each computer provides a different environment for program development, but there are some common elements that deserve comment. No matter what environment you use, you will follow the same basic three steps:

1. Type in a program as a Java class.
2. Compile the program file.
3. Run the compiled version of the program.

The basic unit of storage on most computers is a *file*. Every file has a name. A file name ends with an *extension*, which is the part of a file’s name that follows the period. A file’s extension indicates the type of data contained in the file. For example, files with the extension *.doc* are Microsoft Word documents, and files with the extension *.mp3* are MP3 audio files.



The Java program files that you create must use the extension `.java`. When you compile a Java program, the resulting Java bytecodes are stored in a file with the same name and the extension `.class`.

Most Java programmers use what are known as Integrated Development Environments, or IDEs, which provide an all-in-one environment for creating, editing, compiling, and executing program files. Some of the more popular choices for introductory computer science classes are Eclipse, jGRASP, DrJava, BlueJ, and TextPad. Your instructor will tell you what environment you should use.

Try typing the following simple program in your IDE (the line numbers are not part of the program but are used as an aid):

```
1  public class Hello {  
2      public static void main(String[] args) {  
3          System.out.println("Hello, world!");  
4      }  
5  }
```

Don't worry about the details of this program right now. We will explore those in the next section.

Once you have created your program file, move to step 2 and compile it. The command to compile will be different in each development environment, but the process is the same (typical commands are “compile” or “build”). If any errors are reported, go back to the editor, fix them, and try to compile the program again. (We'll discuss errors in more detail later in this chapter.)

Once you have successfully compiled your program, you are ready to move to step 3, running the program. Again, the command to do this will differ from one environment to the next, but the process is similar (the typical command is “run”). The diagram in Figure 1.1 summarizes the steps you would follow in creating a program called `Hello.java`.

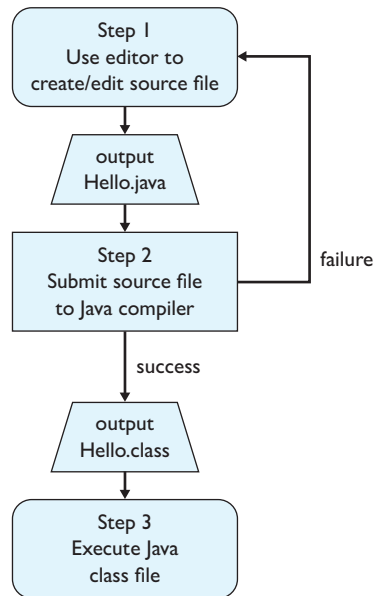
In some IDEs (most notably Eclipse), the first two steps are combined. In these environments the process of compiling is more incremental; the compiler will warn you about errors as you type in code. It is generally not necessary to formally ask such an environment to compile your program because it is compiling as you type.

When your program is executed, it will typically interact with the user in some way. The `Hello.java` program involves an onscreen window known as the *console*.

### Console Window

A special text-only window in which Java programs interact with the user.

The console window is a classic interaction mechanism wherein the computer displays text on the screen and sometimes waits for the user to type responses. This is known as *console* or *terminal interaction*. The text the computer prints to the console window is known as the *output* of the program. Anything typed by the user is known as the console *input*.



**Figure 1.1** Creation and execution of a Java program

To keep things simple, most of the sample programs in this book involve console interaction. Keeping the interaction simple will allow you to focus your attention and effort on other aspects of programming.

## 1.2 And Now—Java

It's time to look at a complete Java program. In the Java programming language, nothing can exist outside of a *class*.

### Class

A unit of code that is the basic building block of Java programs.

The notion of a class is much richer than this, as you'll see when we get to Chapter 8, but for now all you need to know is that each of your Java programs will be stored in a class.

It is a tradition in computer science that when you describe a new programming language, you should start with a program that produces a single line of output with the words, "Hello, world!" The "hello world" tradition has been broken by many authors of Java books because the program turns out not to be as short and simple when it is written in Java as when it is written in other languages, but we'll use it here anyway.

Here is our “hello world” program:

```
1 public class Hello {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, world!");  
4     }  
5 }
```

This program defines a class called `Hello`. Oracle has established the convention that class names always begin with a capital letter, which makes it easy to recognize them. Java requires that the class name and the file name match, so this program must be stored in a file called `Hello.java`. You don’t have to understand all the details of this program just yet, but you do need to understand the basic structure.

The basic form of a Java class is as follows:

```
public class <name> {  
    <method>  
    <method>  
    ...  
    <method>  
}
```

This type of description is known as a *syntax template* because it describes the basic form of a Java construct. Java has rules that determine its legal *syntax* or grammar. Each time we introduce a new element of Java, we’ll begin by looking at its syntax template. By convention, we use the less-than (<) and greater-than (>) characters in a syntax template to indicate items that need to be filled in (in this case, the name of the class and the methods). When we write “...” in a list of elements, we’re indicating that any number of those elements may be included.

The first line of the class is known as the *class header*. The word `public` in the header indicates that this class is available to anyone to use. Notice that the program code in a class is enclosed in curly brace characters (`{ }`). These characters are used in Java to group together related bits of code. In this case, the curly braces are indicating that everything defined within them is part of this public class.

So what exactly can appear inside the curly braces? What can be contained in a class? All sorts of things, but for now, we’ll limit ourselves to *methods*. Methods are the next-smallest unit of code in Java, after classes. A method represents a single action or calculation to be performed.

### Method

A program unit that represents a particular action or computation.

Simple methods are like verbs: They command the computer to perform some action. Inside the curly braces for a class, you can define several different methods.

At a minimum, a complete program requires a special method that is known as the *main* method. It has the following syntax:

```
public static void main(String[] args) {  
    <statement>;  
    <statement>;  
    ...  
    <statement>;  
}
```

Just as the first line of a class is known as a class header, the first line of a method is known as a *method header*. The header for *main* is rather complicated. Most people memorize this as a kind of magical incantation. You want to open the door to Ali Baba's cave? You say, "Open Sesame!" You want to create an executable Java program? You say, `public static void main(String[] args)`. A group of Java teachers make fun of this with a website called [publicstaticvoidmain.com](http://publicstaticvoidmain.com).

Just memorizing magical incantations is never satisfying, especially for computer scientists who like to know everything that is going on in their programs. But this is a place where Java shows its ugly side, and you'll just have to live with it. New programmers, like new drivers, must learn to use something complex without fully understanding how it works. Fortunately, by the time you finish this book, you'll understand every part of the incantation.

Notice that the *main* method has a set of curly braces of its own. They are again used for grouping, indicating that everything that appears between them is part of the *main* method. The lines in between the curly braces specify the series of actions the computer should perform when it executes the method. We refer to these as the *statements* of the method. Just as you put together an essay by stringing together complete sentences, you put together a method by stringing together statements.

### Statement

An executable snippet of code that represents a complete command.

Each statement is terminated by a semicolon. The sample "hello world" program has just a single statement that is known as a `println` statement:

```
System.out.println("Hello, world!");
```

Notice that this statement ends with a semicolon. The semicolon has a special status in Java; it is used to terminate statements in the same way that periods terminate sentences in English.

In the basic "hello world" program there is just a single command to produce a line of output, but consider the following variation (called `He11o2`), which has four lines of code to be executed in the *main* method:

```
1 public class Hello2 {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, world!");  
4         System.out.println();  
5         System.out.println("This program produces four");  
6         System.out.println("lines of output.");  
7     }  
8 }
```

Notice that there are four semicolons in the main method, one at the end of each of the four `println` statements. The statements are executed in the order in which they appear, from first to last, so the `Hello2` program produces the following output:

```
Hello, world!  
  
This program produces four  
lines of output.
```

Let's summarize the different levels we just looked at:

- A Java program is stored in a class.
- Within the class, there are methods. At a minimum, a complete program requires a special method called `main`.
- Inside a method like `main`, there is a series of statements, each of which represents a single command for the computer to execute.

It may seem odd to put the opening curly brace at the end of a line rather than on a line by itself. Some people would use this style of indentation for the program instead:

```
1 public class Hello3  
2 {  
3     public static void main(String[] args)  
4     {  
5         System.out.println("Hello, world!");  
6     }  
7 }
```

Different people will make different choices about the placement of curly braces. The style we use follows Oracle's official Java coding conventions, but the other style has its advocates too. Often people will passionately argue that one way is much better than the other, but it's really a matter of personal taste because each choice has some advantages and some disadvantages. Your instructor may require a particular style; if not, you should choose a style that you are comfortable with and then use it consistently.

Now that you've seen an overview of the structure, let's examine some of the details of Java programs.

**Did You Know?****Hello, World!**

The “hello world” tradition was started by Brian Kernighan and Dennis Ritchie. Ritchie invented a programming language known as C in the 1970s and, together with Kernighan, coauthored the first book describing C, published in 1978. The first complete program in their book was a “hello world” program. Kernighan and Ritchie, as well as their book *The C Programming Language*, have been affectionately referred to as “K & R” ever since.

Many major programming languages have borrowed the basic C syntax as a way to leverage the popularity of C and to encourage programmers to switch to it. The languages C++ and Java both borrow a great deal of their core syntax from C.

Kernighan and Ritchie also had a distinctive style for the placement of curly braces and the indentation of programs that has become known as “K & R style.” This is the style that Oracle recommends and that we use in this book.

**String Literals (Strings)**

When you are writing Java programs (such as the preceding “hello world” program), you’ll often want to include some literal text to send to the console window as output. Programmers have traditionally referred to such text as a *string* because it is composed of a sequence of characters that we string together. The Java language specification uses the term *string literals*.

In Java you specify a string literal by surrounding the literal text in quotation marks, as in

```
"This is a bunch of text surrounded by quotation marks."
```

You must use double quotation marks, not single quotation marks. The following is not a valid string literal:



```
'Bad stuff here.'
```

The following is a valid string literal:

```
"This is a string even with 'these' quotes inside."
```

String literals must not span more than one line of a program. The following is not a valid string literal:



```
"This is really  
bad stuff  
right here."
```

## System.out.println

As you have seen, the `main` method of a Java program contains a series of statements for the computer to carry out. They are executed sequentially, starting with the first statement, then the second, then the third, and so on until the final statement has been executed. One of the simplest and most common statements is `System.out.println`, which is used to produce a line of output. This is another “magical incantation” that you should commit to memory. As of this writing, Google lists around 8,000,000 web pages that mention `System.out.println`. The key thing to remember about this statement is that it’s used to produce a line of output that is sent to the console window.

The simplest form of the `println` statement has nothing inside its parentheses and produces a blank line of output:

```
System.out.println();
```

You need to include the parentheses even if you don’t have anything to put inside them. Notice the semicolon at the end of the line. All statements in Java must be terminated with a semicolon.

More often, however, you use `println` to output a line of text:

```
System.out.println("This line uses the println method.");
```

The above statement commands the computer to produce the following line of output:

```
This line uses the println method.
```

Each `println` statement produces a different line of output. For example, consider the following three statements:

```
System.out.println("This is the first line of output.");  
System.out.println();  
System.out.println("This is the third, below a blank line.");
```

Executing these statements produces the following three lines of output (the second line is blank):

```
This is the first line of output.  
  
This is the third, below a blank line.
```

## Escape Sequences

Any system that involves quoting text will lead you to certain difficult situations. For example, string literals are contained inside quotation marks, so how can you include a quotation mark inside a string literal? String literals also aren’t allowed to break across lines, so how can you include a line break inside a string literal?



The solution is to embed what are known as *escape sequences* in the string literals. Escape sequences are two-character sequences that are used to represent special characters. They all begin with the backslash character (`\`). Table 1.3 lists some of the more common escape sequences.

**Table 1.3 Common Escape Sequences**

Sequence	Represents
<code>\t</code>	tab character
<code>\n</code>	new line character
<code>\"</code>	quotation mark
<code>\\</code>	backslash character

Keep in mind that each of these two-character sequences actually stands for just a single character. For example, consider the following statement:

```
System.out.println("What \"characters\" does this \\ print?");
```

If you executed this statement, you would get the following output:

```
What "characters" does this \ print?
```

The string literal in the `println` has three escape sequences, each of which is two characters long and produces a single character of output.

While string literals themselves cannot span multiple lines (that is, you cannot use a carriage return within a string literal to force a line break), you can use the `\n` escape sequence to embed new line characters in a string. This leads to the odd situation where a single `println` statement can produce more than one line of output.

For example, consider this statement:

```
System.out.println("This\nproduces 3 lines\nof output.");
```

If you execute it, you will get the following output:

```
This
produces 3 lines
of output.
```

The `println` itself produces one line of output, but the string literal contains two new line characters that cause it to be broken up into a total of three lines of output. To produce the same output without new line characters, you would have to issue three separate `println` statements.

This is another programming habit that tends to vary according to taste. Some people (including the authors) find it hard to read string literals that contain `\n` escape sequences, but other people prefer to write fewer lines of code. Once again, you should make up your own mind about when to use the new line escape sequence.

## print versus println

Java has a variation of the `println` command called `print` that allows you to produce output on the current line without going to a new line of output. The `println` command really does two different things: It sends output to the current line, and then it moves to the beginning of a new line. The `print` command does only the first of these. Thus, a series of `print` commands will generate output all on the same line. Only a `println` command will cause the current line to be completed and a new line to be started. For example, consider these six statements:

```
System.out.print("To be ");
System.out.print("or not to be.");
System.out.print("That is ");
System.out.println("the question.");
System.out.print("This is");
System.out.println(" for the whole family!");
```

These statements produce two lines of output. Remember that every `println` statement produces exactly one line of output; because there are two `println` statements here, there are two lines of output. After the first statement executes, the current line looks like this:

```
To be ^
```

The arrow below the output line indicates the position where output will be sent next. We can simplify our discussion if we refer to the arrow as the *output cursor*. Notice that the output cursor is at the end of this line and that it appears after a space. The reason is that the command was a `print` (doesn't go to a new line) and the string literal in the `print` ended with a space. Java will not insert a space for you unless you specifically request it. After the next `print`, the line looks like this:

```
To be or not to be. ^
```

There's no space at the end now because the string literal in the second `print` command ends in a period, not a space. After the next `print`, the line looks like this:

```
To be or not to be.That is ^
```

There is no space between the period and the word "That" because there was no space in the `print` commands, but there is a space at the end of the string literal in the third statement. After the next statement executes, the output looks like this:

```
To be or not to be.That is the question.
```

```
^
```

Because this fourth statement is a `println` command, it finishes the output line and positions the cursor at the beginning of the second line. The next statement is another `print` that produces this:

```
To be or not to be.That is the question.
This is
^
```

The final `println` completes the second line and positions the output cursor at the beginning of a new line:

```
To be or not to be.That is the question.
This is for the whole family!

^
```

These six statements are equivalent to the following two single statements:

```
System.out.println("To be or not to be.That is the question.");
System.out.println("This is for the whole family!");
```

Using the `print` and `println` commands together to produce lines like these may seem a bit silly, but you will see that there are more interesting applications of `print` in the next chapter.

Remember that it is possible to have an empty `println` command:

```
System.out.println();
```

Because there is nothing inside the parentheses to be written to the output line, this command positions the output cursor at the beginning of the next line. If there are `print` commands before this empty `println`, it finishes out the line made by those `print` commands. If there are no previous `print` commands, it produces a blank line. An empty `print` command is meaningless and illegal.

## Identifiers and Keywords

The words used to name parts of a Java program are called *identifiers*.

### Identifier

A name given to an entity in a program, such as a class or method.

Identifiers must start with a letter, which can be followed by any number of letters or digits. The following are all legal identifiers:

```
first           hiThere       numStudents    TwoBy4
```

The Java language specification defines the set of letters to include the underscore and dollar-sign characters (`_` and `$`), which means that the following are legal identifiers as well:

```
two_plus_two      _count      $2donuts      MAX_COUNT
```

The following are illegal identifiers:



```
two+two      hi there      hi-There      2by4
```

Java has conventions for capitalization that are followed fairly consistently by programmers. All class names should begin with a capital letter, as with the `Hello`, `Hello2`, and `Hello3` classes introduced earlier. The names of methods should begin with lowercase letters, as in the `main` method. When you are putting several words together to form a class or method name, capitalize the first letter of each word after the first. In the next chapter we'll discuss constants, which have yet another capitalization scheme, with all letters in uppercase and words separated by underscores. These different schemes might seem like tedious constraints, but using consistent capitalization in your code allows the reader to quickly identify the various code elements.

For example, suppose that you were going to put together the words “all my children” into an identifier. The result would be

- `AllMyChildren` for a class name (each word starts with a capital)
- `allMyChildren` for a method name (starts with a lowercase letter, subsequent words capitalized)
- `ALL_MY_CHILDREN` for a constant name (all uppercase, with words separated by underscores; described in Chapter 2)

Java is case sensitive, so the identifiers `class`, `Class`, `CLASS`, and `clAss` are all considered different. Keep this in mind as you read error messages from the compiler. People are good at understanding what you write, even if you misspell words or make little mistakes like changing the capitalization of a word. However, mistakes like these cause the Java compiler to become hopelessly confused.

Don't hesitate to use long identifiers. The more descriptive your names are, the easier it will be for people (including you) to read your programs. Descriptive identifiers are worth the time they take to type. Java's `String` class, for example, has a method called `compareToIgnoreCase`.

Be aware, however, that Java has a set of predefined identifiers called *keywords* that are reserved for particular uses. As you read this book, you will learn many of these keywords and their uses. You can only use keywords for their intended purposes. You must be careful to avoid using these words in the names of identifiers. For example, if you name a method `short` or `try`, this will cause a problem, because `short` and `try` are reserved keywords. Table 1.4 shows the complete list of reserved keywords.

**Table 1.4 List of Java Keywords**

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

### A Complex Example: DrawFigures1

The `println` statement can be used to draw text figures as output. Consider the following more complicated program example (notice that it uses two empty `println` statements to produce blank lines):

```

1  public class DrawFigures1 {
2      public static void main(String[] args) {
3          System.out.println("  /\");
4          System.out.println(" /  \");
5          System.out.println("/    \");
6          System.out.println("\ \  /");
7          System.out.println("\ \ /");
8          System.out.println("  \/");
9          System.out.println();
10         System.out.println("\ \  /");
11         System.out.println("\ \ /");
12         System.out.println("  \/");
13         System.out.println("  /\");
14         System.out.println(" /  \");
15         System.out.println("/    \");
16         System.out.println();
17         System.out.println("  /\");
18         System.out.println(" /  \");
19         System.out.println("/    \");
20         System.out.println("+-----+");
21         System.out.println("|          |");
22         System.out.println("|          |");
23         System.out.println("+-----+");
24         System.out.println("|United|");
25         System.out.println("|States|");
26         System.out.println("+-----+");
27         System.out.println("|          |");


```

```

28         System.out.println(" |      |");
29         System.out.println("+-----+");
30         System.out.println("    /\\"");
31         System.out.println("  /  \\"");
32         System.out.println(" /    \\"");
33     }
34 }

```

The following is the output the program generates. Notice that the program includes double backslash characters (\\), but the output has single backslash characters. This is an example of an escape sequence, as described previously.



```

  /\
 /\ 
 /\ 
 /\ 
  /\

 \  /
 \  /
 \  /
 \  /
 \  /

 /\
 /\ 
 /\ 
+-----+
|       |
|       |
+-----+
|United |
|States |
+-----+
|       |
|       |
+-----+

 /\
 /\ 
 /\ 

```

## Comments and Readability

Java is a free-format language. This means you can put in as many or as few spaces and blank lines as you like, as long as you put at least one space or other punctuation mark between words. However, you should bear in mind that the layout of a program can enhance (or detract from) its readability. The following program is legal but hard to read:

```

1  public class Ugly{public static void main(String[] args)
2  {System.out.println("How short I am!");}}
```

Here are some simple rules to follow that will make your programs more readable:

- Put class and method headers on lines by themselves.
- Put no more than one statement on each line.
- Indent your program properly. When an opening brace appears, increase the indentation of the lines that follow it. When a closing brace appears, reduce the indentation. Indent statements inside curly braces by a consistent number of spaces (a common choice is four spaces per level of indentation).
- Use blank lines to separate parts of the program (e.g., methods).

Using these rules to rewrite the `Ugly` program yields the following code:

```
1 public class Ugly {
2     public static void main(String[] args) {
3         System.out.println("How short I am!");
4     }
5 }
```

Well-written Java programs can be quite readable, but often you will want to include some explanations that are not part of the program itself. You can annotate programs by putting notes called *comments* in them.


### Comment

Text that programmers include in a program to explain their code. The compiler ignores comments.

There are two comment forms in Java. In the first form, you open the comment with a slash followed by an asterisk and you close it with an asterisk followed by a slash:

```
/* like this */
```

You must not put spaces between the slashes and the asterisks:

 `/ * this is bad * /`

You can put almost any text you like, including multiple lines, inside the comment:

```
/* Thaddeus Martin
   Assignment #1
   Instructor: Professor Walingford
   Grader:     Bianca Montgomery */
```

The only things you aren't allowed to put inside a comment are the comment end characters. The following code is not legal:



```
/* This comment has an asterisk/slash /*/ in it,  
   which prematurely closes the comment. This is bad. */
```

Java also provides a second comment form for shorter, single-line comments. You can use two slashes in a row to indicate that the rest of the current line (everything to the right of the two slashes) is a comment. For example, you can put a comment after a statement:

```
System.out.println("You win!"); // Good job!
```

Or you can create a comment on its own line:

```
// give an introduction to the user  
System.out.println("Welcome to the game of blackjack.");  
System.out.println();  
System.out.println("Let me explain the rules.");
```

You can even create blocks of single-line comments:

```
// Thaddeus Martin  
// Assignment #1  
// Instructor:  Professor Walingford  
// Grader:      Bianca Montgomery
```

Some people prefer to use the first comment form for comments that span multiple lines but it is safer to use the second form because you don't have to remember to close the comment. It also makes the comment stand out more. This is another case in which, if your instructor does not tell you to use a particular comment style, you should decide for yourself which style you prefer and use it consistently.

Don't confuse comments with the text of `println` statements. The text of your comments will not be displayed as output when the program executes. The comments are there only to help readers examine and understand the program.

It is a good idea to include comments at the beginning of each class file to indicate what the class does. You might also want to include information about who you are, what course you are taking, your instructor and/or grader's name, the date, and so on. You should also comment each method to indicate what it does.

Commenting becomes more useful in larger and more complicated programs, as well as in programs that will be viewed or modified by more than one programmer. Clear comments are extremely helpful to explain to another person, or to yourself at a later time, what your program is doing and why it is doing it.

In addition to the two comment forms already discussed, Java supports a particular style of comments known as *Javadoc comments*. Their format is more complex, but they have the advantage that you can use a program to extract the comments to make HTML files suitable for reading with a web browser. Javadoc comments are useful in more advanced programming and are discussed in more detail in Appendix B.



## 1.3 Program Errors

In 1949, Maurice Wilkes, an early pioneer of computing, expressed a sentiment that still rings true today:

As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

You also will have to face this reality as you learn to program. You're going to make mistakes, just like every other programmer in history, and you're going to need strategies for eliminating those mistakes. Fortunately, the computer itself can help you with some of the work.

There are three kinds of errors that you'll encounter as you write programs:

- *Syntax errors* occur when you misuse Java. They are the programming equivalent of bad grammar and are caught by the Java compiler.
- *Logic errors* occur when you write code that doesn't perform the task it is intended to perform.
- *Runtime errors* are logic errors that are so severe that Java stops your program from executing.

### Syntax Errors

Human beings tend to be fairly forgiving about minor mistakes in speech. For example, we might find Master Yoda's phrasing odd, but we generally have no problems in understanding him.

The Java compiler will be far less forgiving. The compiler reports syntax errors as it attempts to translate your program from Java into bytecodes if your program breaks any of Java's grammar rules. For example, if you misplace a single semicolon in your program, you can send the compiler into a tailspin of confusion. The compiler may report several error messages, depending on what it thinks is wrong with your program.

A program that generates compilation errors cannot be executed. If you submit your program to the compiler and the compiler reports errors, you must fix the errors and resubmit the program. You will not be able to proceed until your program is free of compilation errors.

Some development environments, such as Eclipse, help you along the way by underlining syntax errors as you write your program. This makes it easy to spot exactly where errors occur.

It's possible for you to introduce an error before you even start writing your program, if you choose the wrong name for its file.

### Common Programming Error

#### File Name Does Not Match Class Name

As mentioned earlier, Java requires that a program's class name and file name match. For example, a program that begins with `public class Hello` must be stored in a file called `Hello.java`.

If you use the wrong file name (for example, saving it as `WrongFileName.java`), you'll get an error message like this:

```
WrongFileName.java:1: error: class Hello is public,  
    should be declared in a file named Hello.java  
public class Hello {  
    ^  
1 error
```

The file name is just the first hurdle. A number of other errors may exist in your Java program. One of the most common syntax errors is to misspell a word. You may have punctuation errors, such as missing semicolons. It's also easy to forget an entire word, such as a required keyword.

The error messages the compiler gives may or may not be helpful. If you don't understand the content of the error message, look for the caret marker (^) below the line, which points at the position in the line where the compiler became confused. This can help you pinpoint the place where a required keyword might be missing.

### Common Programming Error

#### Misspelled Words

Java (like most programming languages) is very picky about spelling. You need to spell each word correctly, including proper capitalization. Suppose, for example, that you were to replace the `println` statement in the "hello world" program with the following:



```
System.out.pruntln("Hello, world!");
```

When you try to compile this program, it will generate an error message similar to the following:

```
Hello.java:3: error: cannot find symbol  
symbol   : method pruntln(java.lang.String)
```

*Continued on next page*

*Continued from previous page*

```
location: variable out of type PrintStream
      System.out.pruntln("Hello, world!");
              ^
1 error
```

The first line of this output indicates that the error occurs in the file `Hello.java` on line 3 and that the error is that the compiler cannot find a symbol. The second line indicates that the symbol it can't find is a method called `pruntln`. That's because there is no such method; the method is called `println`. The error message can take slightly different forms depending on what you have misspelled. For example, you might forget to capitalize the word `System`:



```
system.out.println("Hello, world!");
```

You will get the following error message:

```
Hello.java:3: error: package system does not exist
      system.out.println("Hello, world!");
              ^
1 error
```

Again, the first line indicates that the error occurs in line 3 of the file `Hello.java`. The error message is slightly different here, though, indicating that it can't find a package called `system`. The second and third lines of this error message include the original line of code with an arrow (caret) pointing to where the compiler got confused. The compiler errors are not always very clear, but if you pay attention to where the arrow is pointing, you'll have a pretty good sense of where the error occurs.

If you still can't figure out the error, try looking at the error's line number and comparing the contents of that line with similar lines in other programs. You can also ask someone else, such as an instructor or lab assistant, to examine your program.

### Common Programming Error

#### Forgetting a Semicolon

All Java statements must end with semicolons, but it's easy to forget to put a semicolon at the end of a statement, as in the following program:



```
1 public class MissingSemicolon {
2     public static void main(String[] args) {
3         System.out.println("A rose by any other name")
```

*Continued on next page*

*Continued from previous page*

```
4      System.out.println("would smell as sweet");
5  }
6 }
```

In this case, the compiler produces output similar to the following:

```
MissingSemicolon.java:3: error: ';' expected
    System.out.println("would smell as sweet");
    ^
1 error
```

Some versions of the Java compiler list line 4 as the cause of the problem, not line 3, where the semicolon was actually forgotten. This is because the compiler is looking forward for a semicolon and isn't upset until it finds something that isn't a semicolon, which it does when it reaches line 4. Unfortunately, as this case demonstrates, compiler error messages don't always direct you to the correct line to be fixed.

## Common Programming Error

### Forgetting a Required Keyword

Another common syntax error is to forget a required keyword when you are typing your program, such as `static` or `class`. Double-check your programs against the examples in the textbook to make sure you haven't omitted an important keyword.

The compiler will give different error messages depending on which keyword is missing, but the messages can be hard to understand. For example, you might write a program called `Bug4` and forget the keyword `class` when writing its class header. In this case, the compiler will provide the following error message:

```
Bug4.java:1: error: class, interface, or enum expected
public Bug4 {
    ^
1 error
```

However, if you forget the keyword `void` when declaring the main method, the compiler generates a different error message:

```
Bug5.java:2: error: invalid method declaration; return type required
    public static main(String[] args) {
    ^
1 error
```

Yet another common syntax error is to forget to close a string literal.

A good rule of thumb to follow is that the first error reported by the compiler is the most important one. The rest might be the result of that first error. Many programmers don't even bother to look at errors beyond the first, because fixing that error and recompiling may cause the other errors to disappear.

## Logic Errors (Bugs)

Logic errors are also called *bugs*. Computer programmers use words like “bug-ridden” and “buggy” to describe poorly written programs, and the process of finding and eliminating bugs from programs is called *debugging*.

The word “bug” is an old engineering term that predates computers; early computing bugs sometimes occurred in hardware as well as software. Admiral Grace Hopper, an early pioneer of computing, is largely credited with popularizing the use of the term in the context of computer programming. She often told the true story of a group of programmers at Harvard University in the mid-1940s who couldn't figure out what was wrong with their programs until they opened up the computer and found an actual moth trapped inside.

The form that a bug takes may vary. Sometimes your program will simply behave improperly. For example, it might produce the wrong output. Other times it will ask the computer to perform some task that is clearly a mistake, in which case your program will have a runtime error that stops it from executing. In this chapter, since your knowledge of Java is limited, generally the only type of logic error you will see is a mistake in program output from an incorrect `println` statement or method call.

We'll look at an example of a runtime error in the next section.

## 1.4 Procedural Decomposition

Brian Kernighan, coauthor of *The C Programming Language*, has said, “Controlling complexity is the essence of computer programming.” People have only a modest capacity for detail. We can't solve complex problems all at once. Instead, we structure our problem solving by dividing the problem into manageable pieces and conquering each piece individually. We often use the term *decomposition* to describe this principle as applied to programming.

### Decomposition

A separation into discernible parts, each of which is simpler than the whole.

With procedural programming languages like C, decomposition involves dividing a complex task into a set of subtasks. This is a very verb- or action-oriented approach, involving dividing up the overall action into a series of smaller actions. This technique is called *procedural decomposition*.

**Common Programming Error****Not Closing a String Literal or Comment**

Every string literal has to have an opening quote and a closing quote, but it's easy to forget the closing quotation mark. For example, you might say:



```
System.out.println("Hello, world!);
```

This produces three different error messages, even though there is only one underlying syntax error:

```
Hello.java:3: error: unclosed string literal
    System.out.println("hello world);
                        ^
Hello.java:3: error: ';' expected
    System.out.println("hello world);
                        ^
Hello.java:5: error: reached end of file while parsing
    }
    ^
3 errors
```

In this case, the first error message is quite clear, including an arrow pointing at the beginning of the string literal that wasn't closed. The second error message was caused by the first. Because the string literal was not closed, the compiler didn't notice the right parenthesis and semicolon that appear at the end of the line.

A similar problem occurs when you forget to close a multiline comment by writing `*/`, as in the first line of the following program:



```
/* This is a bad program.

public class Bad {
    public static void main(String[] args){
        System.out.println("Hi there.");
    }
} /* end of program */
```

The preceding file is not a program; it is one long comment. Because the comment on the first line is not closed, the entire program is swallowed up.

Luckily, many Java editor programs color the parts of a program to help you identify them visually. Usually, if you forget to close a string literal or comment, the rest of your program will turn the wrong color, which can help you spot the mistake.

Java was designed for a different kind of decomposition that is more noun- or object-oriented. Instead of thinking of the problem as a series of actions to be performed, we think of it as a collection of objects that have to interact.

As a computer scientist, you should be familiar with both types of problem solving. This book begins with procedural decomposition and devotes many chapters to mastering various aspects of the procedural approach. Only after you have thoroughly practiced procedural programming will we turn our attention back to object decomposition and object-oriented programming.

As an example of procedural decomposition, consider the problem of baking a cake. You can divide this problem into the following subproblems:

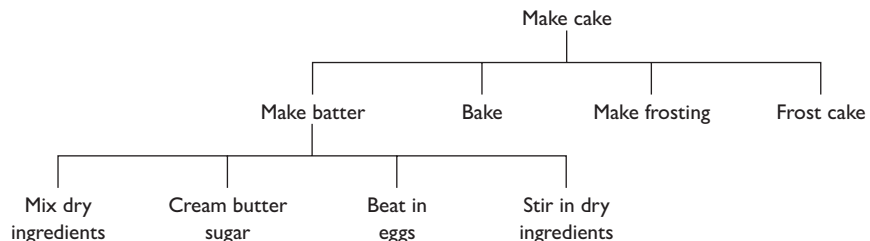
- Make the batter.
- Bake the cake.
- Make the frosting.
- Frost the cake.

Each of these four tasks has details associated with it. To make the batter, for example, you follow these steps:

- Mix the dry ingredients.
- Cream the butter and sugar.
- Beat in the eggs.
- Stir in the dry ingredients.

Thus, you divide the overall task into subtasks, which you further divide into even smaller subtasks. Eventually, you reach descriptions that are so simple they require no further explanation (i.e., primitives).

A partial diagram of this decomposition is shown in Figure 1.2. “Make cake” is the highest-level operation. It is defined in terms of four lower-level operations called “Make batter,” “Bake,” “Make frosting,” and “Frost cake.” The “Make batter” operation is defined in terms of even lower-level operations, and the same could be done for the other three operations. This diagram is called a structure diagram and is intended to show how a problem is broken down into subproblems. In this diagram, you can also tell in what order operations are performed by reading from left to right. That is not true of most structure diagrams. To determine the actual order in which subprograms are performed, you usually have to refer to the program itself.



**Figure 1.2** Decomposition of “Make cake” task

One final problem-solving term has to do with the process of programming. Professional programmers develop programs in stages. Instead of trying to produce a complete working program all at once, they choose some piece of the problem to implement first. Then they add another piece, and another, and another. The overall program is built up slowly, piece by piece. This process is known as *iterative enhancement* or *stepwise refinement*.

### Iterative Enhancement

The process of producing a program in stages, adding new functionality at each stage. A key feature of each iterative step is that you can test it to make sure that piece works before moving on.

Now, let's look at a construct that will allow you to iteratively enhance your Java programs to improve their structure and reduce their redundancy: static methods.

## Static Methods



Java is designed for objects, and programming in Java usually involves decomposing a problem into various objects, each with methods that perform particular tasks. You will see how this works in later chapters, but for now, we are going to explore procedural decomposition. We will postpone examining some of Java's details while we discuss programming in general.

Consider the following program, which draws two text boxes on the console:

```
1 public class DrawBoxes {
2     public static void main(String[] args) {
3         System.out.println("+-----+");
4         System.out.println("|         |");
5         System.out.println("|         |");
6         System.out.println("+-----+");
7         System.out.println();
8         System.out.println("+-----+");
9         System.out.println("|         |");
10        System.out.println("|         |");
11        System.out.println("+-----+");
12    }
13 }
```

The program works correctly, but the four lines used to draw the box appear twice. This redundancy is undesirable for several reasons. For example, you might wish to change the appearance of the boxes, in which case you'll have to make all of the edits twice. Also, you might wish to draw additional boxes, which would require you to type additional copies of (or copy and paste) the redundant lines.



A preferable program would include a Java command that specifies how to draw the box and then executes that command twice. Java doesn't have a "draw a box" command, but you can create one. Such a named command is called a *static method*.

### Static Method

A block of Java statements that is given a name.

Static methods are units of procedural decomposition. We typically break a class into several static methods, each of which solves some piece of the overall problem. For example, here is a static method to draw a box:

```
public static void drawBox() {  
    System.out.println("+-----+");  
    System.out.println("|         |");  
    System.out.println("|         |");  
    System.out.println("+-----+");  
}
```

You have already seen a static method called `main` in earlier programs. Recall that the `main` method has the following form:

```
public static void main(String[] args) {  
    <statement>;  
    <statement>;  
    ...  
    <statement>;  
}
```

The static methods you'll write have a similar structure:

```
public static void <name>() {  
    <statement>;  
    <statement>;  
    ...  
    <statement>;  
}
```

The first line is known as the method header. You don't yet need to fully understand what each part of this header means in Java; for now, just remember that you'll need to write `public static void`, followed by the name you wish to give the method, followed by a set of parentheses. Briefly, here is what the words in the header mean:

- The keyword `public` indicates that this method is available to be used by all parts of your program. All methods you write will be `public`.
- The keyword `static` indicates that this is a static (procedural-style, not object-oriented) method. For now, all methods you write will be static, until you learn about defining objects in Chapter 8.
- The keyword `void` indicates that this method executes statements but does not produce any value. (Other methods you'll see later compute and return values.)
- `<name>` (e.g., `drawBox`) is the name of the method.
- The empty parentheses specify a list (in this case, an empty list) of values that are sent to your method as input; such values are called *parameters* and will not be included in your methods until Chapter 3.

Including the keyword `static` for each method you define may seem cumbersome. Other Java textbooks often do not discuss static methods as early as we do here; instead, they show other techniques for decomposing problems. But even though static methods require a bit of work to create, they are powerful and useful tools for improving basic Java programs.

After the header in our sample method, a series of `println` statements makes up the body of this static method. As in the `main` method, the statements of this method are executed in order from first to last.

By defining the method `drawBox`, you have given a simple name to this sequence of `println` statements. It's like saying to the Java compiler, "Whenever I tell you to 'drawBox,' I really mean that you should execute the `println` statements in the `drawBox` method." But the command won't actually be executed unless our `main` method explicitly says that it wants to do so. The act of executing a static method is called a *method call*.

#### Method Call

A command to execute another method, which causes all of the statements inside that method to be executed.

To execute the `drawBox` command, include this line in your program's `main` method:

```
drawBox();
```

Since we want to execute the `drawBox` command twice (to draw two boxes), the `main` method should contain two calls to the `drawBox` method. The following

program uses the `drawBox` method to produce the same output as the original `DrawBoxes` program:

```

1  public class DrawBoxes2 {
2      public static void main(String[] args) {
3          drawBox();
4          System.out.println();
5          drawBox();
6      }
7
8      public static void drawBox() {
9          System.out.println("+-----+");
10         System.out.println("|         |");
11         System.out.println("|         |");
12         System.out.println("+-----+");
13     }
14 }

```

## Flow of Control

The most confusing thing about static methods is that programs with static methods do not execute sequentially from top to bottom. Rather, each time the program encounters a static method call, the execution of the program “jumps” to that static method, executes each statement in that method in order, and then “jumps” back to the point where the call began and resumes executing. The order in which the statements of a program are executed is called the program’s *flow of control*.

### Flow of Control

The order in which the statements of a Java program are executed.

Let’s look at the control flow of the `DrawBoxes2` program shown previously. It has two methods. The first method is the familiar `main` method, and the second is `drawBox`. As in any Java program, execution starts with the `main` method:

```

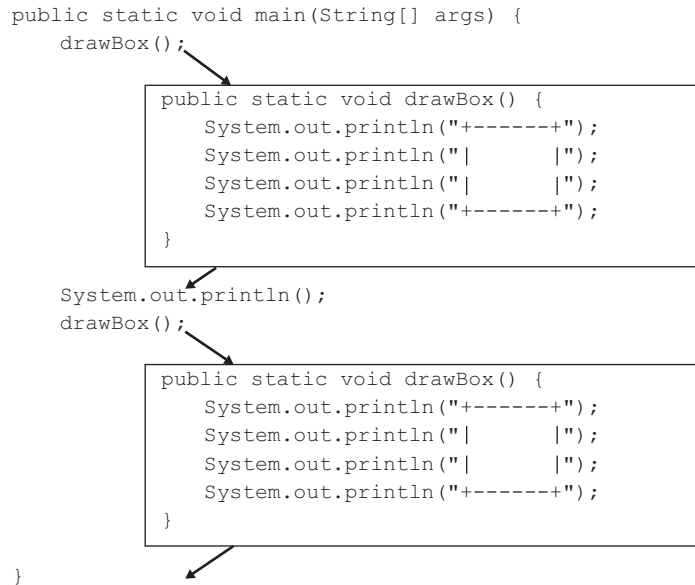
public static void main(String[] args) {
    drawBox();
    System.out.println();
    drawBox();
}

```

In a sense, the execution of this program is sequential: Each statement listed in the `main` method is executed in turn, from first to last.

But this `main` method includes two different calls on the `drawBox` method. This program will do three different things: execute `drawBox`, execute a `println`, then execute `drawBox` again.

The diagram below indicates the flow of control produced by this program.



Following the diagram, you can see that nine `println` statements are executed. First you transfer control to the `drawBox` method and execute its four statements. Then you return to `main` and execute its `println` statement. Then you transfer control a second time to `drawBox` and once again execute its four statements. Making these method calls is almost like copying and pasting the code of the method into the `main` method. As a result, this program has the exact same behavior as the nine-line `main` method of the `DrawBoxes` program:

```

public static void main(String[] args) {
    System.out.println("+-----+");
    System.out.println("|         |");
    System.out.println("|         |");
    System.out.println("+-----+");
    System.out.println();
    System.out.println("+-----+");
    System.out.println("|         |");
    System.out.println("|         |");
    System.out.println("+-----+");
}

```

This version is simpler in terms of its flow of control, but the first version avoids the redundancy of having the same `println` statements appear multiple times. It also gives a better sense of the structure of the solution. In the original version, it is clear that there is a subtask called `drawBox` that is being performed twice. Also, while the last version of

the main method contains fewer lines of code than the `DrawBoxes2` program, consider what would happen if you wanted to add a third box to the output. You would have to add the five requisite `println` statements again, whereas in the programs that use the `drawBox` method you can simply add one more `println` and a third method call.

Java allows you to define methods in any order you like. It is a common convention to put the main method as either the first or last method in the class. In this textbook we will generally put main first, but the programs would behave the same if we switched the order. For example, the following modified program behaves identically to the previous `DrawBoxes2` program:

```

1  public class DrawBoxes3 {
2      public static void drawBox() {
3          System.out.println("+-----+");
4          System.out.println("|         |");
5          System.out.println("|         |");
6          System.out.println("+-----+");
7      }
8
9      public static void main(String[] args) {
10         drawBox();
11         System.out.println();
12         drawBox();
13     }
14 }
```

The main method is always the starting point for program execution, and from that starting point you can determine the order in which other methods are called.

## Methods That Call Other Methods

The main method is not the only place where you can call another method. In fact, any method may call any other method. As a result, the flow of control can get quite complicated. Consider, for example, the following rather strange program. We use nonsense words (“foo,” “bar,” “baz,” and “mumble”) on purpose because the program is not intended to make sense.

```

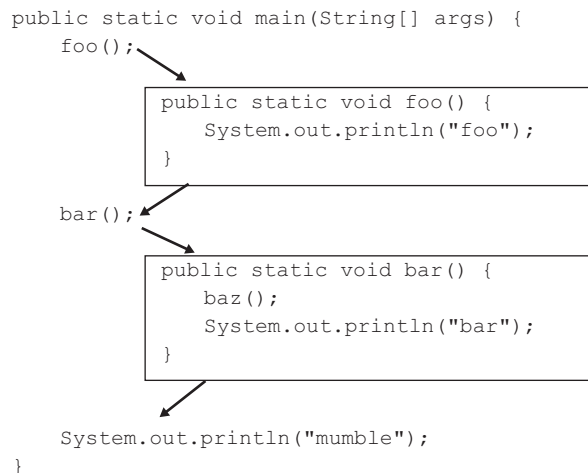
1  public class FooBarBazMumble {
2      public static void main(String[] args) {
3          foo();
4          bar();
5          System.out.println("mumble");
6      }
7
8      public static void foo() {
9          System.out.println("foo");
10     }
```

```
11
12     public static void bar() {
13         baz();
14         System.out.println("bar");
15     }
16
17     public static void baz() {
18         System.out.println("baz");
19     }
20 }
```

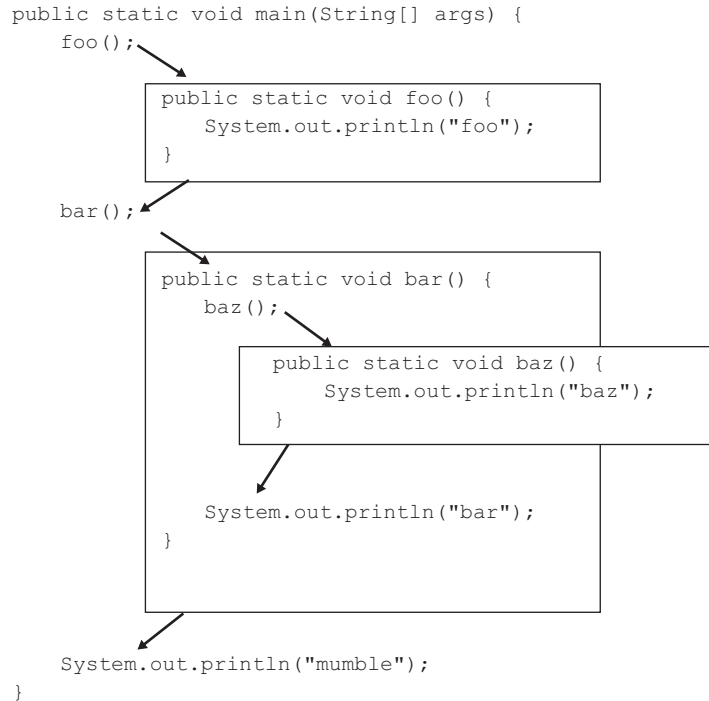
You can't tell easily what output this program produces, so let's explore in detail what the program is doing. Remember that Java always begins with the method called `main`. In this program, the `main` method calls the `foo` method and the `bar` method and then executes a `println` statement:

```
public static void main(String[] args) {
    foo();
    bar();
    System.out.println("mumble");
}
```

Each of these two method calls will expand into more statements. Let's first expand the calls on the `foo` and `bar` methods:



This helps to make our picture of the flow of control more complete, but notice that `bar` calls the `baz` method, so we have to expand that as well.



Finally, we have finished our picture of the flow of control of this program. It should make sense, then, that the program produces the following output:

```

foo
baz
bar
mumble

```

We will see a much more useful example of methods calling methods when we go through the case study at the end of the chapter.

### Did You Know?

#### *The New Hacker's Dictionary*

Computer scientists and computer programmers use a lot of jargon that can be confusing to novices. A group of software professionals spearheaded by Eric Raymond have collected together many of the jargon terms in a book called *The New Hacker's Dictionary*. You can buy the book, or you can browse it online at Eric's website: <http://catb.org/esr/jargon/html/frames.html>.

For example, if you look up *foo*, you'll find this definition: "Used very generally as a sample name for absolutely anything, esp. programs and files." In

*Continued on next page*

*Continued from previous page*

other words, when we find ourselves looking for a nonsense word, we use “foo.”

*The New Hacker’s Dictionary* contains a great deal of historical information about the origins of jargon terms. The entry for *foo* includes a lengthy discussion of the combined term *foobar* and how it came into common usage among engineers.

If you want to get a flavor of what is there, check out the entries for *bug*, *hacker*, *bogosity*, and *bogo-sort*.

## An Example Runtime Error

Runtime errors occur when a bug causes your program to be unable to continue executing. What could cause such a thing to happen? One example is if you asked the computer to calculate an invalid value, such as 1 divided by 0. Another example would be if your program tries to read data from a file that does not exist.

We haven’t discussed how to compute values or read files yet, but there is a way you can “accidentally” cause a runtime error. The way to do this is to write a static method that calls itself. If you do this, your program will not stop running, because the method will keep calling itself indefinitely, until the computer runs out of memory. When this happens, the program prints a large number of lines of output, and then eventually stops executing with an error message called a `StackOverflowError`. Here’s an example:

```
1 public class Infinite {
2     public static void main(String[] args) {
3         oops();
4     }
5
6     public static void oops() {
7         System.out.println("Make it stop!");
8         oops();
9     }
10 }
```

This ill-fated program produces the following output (with large groups of identical lines represented by “...”):

```
Make it stop!
Make it stop!
Make it stop!
Make it stop!
Make it stop!
Make it stop!
Make it stop!
```



```

Make it stop!
Make it stop!
...
Make it stop!
Exception in thread "main" java.lang.StackOverflowError
    at sun.nio.cs.SingleByteEncoder.encodeArrayLoop(Unknown Source)
    at sun.nio.cs.SingleByteEncoder.encodeLoop(Unknown Source)
    at java.nio.charset.CharsetEncoder.encode(Unknown Source)
    at sun.nio.cs.StreamEncoder$CharsetSE.implWrite(Unknown Source)
    at sun.nio.cs.StreamEncoder.write(Unknown Source)
    at java.io.OutputStreamWriter.write(Unknown Source)
    at java.io.BufferedWriter.flushBuffer(Unknown Source)
    at java.io.PrintStream.newLine(Unknown Source)
    at java.io.PrintStream.println(Unknown Source)
    at Infinite.oops(Infinite.java:7)
    at Infinite.oops(Infinite.java:8)
    at Infinite.oops(Infinite.java:8)
    at Infinite.oops(Infinite.java:8)
    at ...

```

Runtime errors are, unfortunately, something you'll have to live with as you learn to program. You will have to carefully ensure that your programs not only compile successfully, but do not contain any bugs that will cause a runtime error. The most common way to catch and fix runtime errors is to run the program several times to test its behavior.

## 1.5 Case Study: DrawFigures



VideoNote

Earlier in the chapter, you saw a program called `DrawFigures1` that produced the following output:

```

  /\
 /\
/\
 \/\
  /\

  /\
 /\
/\
 \/\
  /\

```



It did so with a long sequence of `println` statements in the `main` method. In this section you'll improve the program by using static methods for procedural decomposition to capture structure and eliminate redundancy. The redundancy might be more obvious, but let's start by improving the way the program captures the structure of the overall task.

### Structured Version

If you look closely at the output, you'll see that it has a structure that would be desirable to capture in the program structure. The output is divided into three subfigures: the diamond, the X, and the rocket.

You can better indicate the structure of the program by dividing it into static methods. Since there are three subfigures, you can create three methods, one for each subfigure. The following program produces the same output as `DrawFigures1`:

```

1  public class DrawFigures2 {
2      public static void main(String[] args) {
3          drawDiamond();
4          drawX();
5          drawRocket();
6      }
7
8      public static void drawDiamond() {
9          System.out.println("  /\\"");
10         System.out.println(" /  \\\");
11         System.out.println("/    \\\");
12         System.out.println("\\    /");
13         System.out.println("\\  /");
14         System.out.println("\\ /");
15         System.out.println();
16     }
17

```

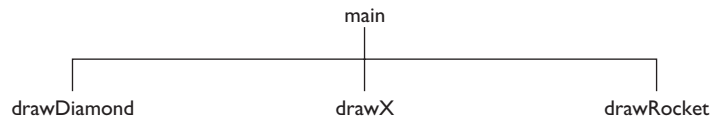
```

18     public static void drawX() {
19         System.out.println("  \\    /");
20         System.out.println("   \\  /");
21         System.out.println("    \\ /");
22         System.out.println("     /\\"");
23         System.out.println("    /  \\"");
24         System.out.println("   /    \\"");
25         System.out.println();
26     }
27
28     public static void drawRocket() {
29         System.out.println("     /\\"");
30         System.out.println("    /  \\"");
31         System.out.println("   /    \\"");
32         System.out.println("+-----+");
33         System.out.println("|        |");
34         System.out.println("|        |");
35         System.out.println("+-----+");
36         System.out.println("|United|");
37         System.out.println("|States|");
38         System.out.println("+-----+");
39         System.out.println("|        |");
40         System.out.println("|        |");
41         System.out.println("+-----+");
42         System.out.println("     /\\"");
43         System.out.println("    /  \\"");
44         System.out.println("   /    \\"");
45     }
46 }

```

The program appears in a class called `DrawFigures2` and has four static methods defined within it. The first static method is the usual `main` method, which calls three methods. The three methods called by `main` appear next.

Figure 1.3 is a structure diagram for this version of the program. Notice that it has two levels of structure. The overall problem is broken down into three subtasks.



**Figure 1.3** Decomposition of `DrawFigures2`

## Final Version without Redundancy

The program can still be improved. Each of the three subfigures has individual elements, and some of those elements appear in more than one of the three subfigures. The program prints the following redundant group of lines several times:



A better version of the preceding program adds an additional method for each redundant section of output. The redundant sections are the top and bottom halves of the diamond shape and the box used in the rocket. Here is the improved program:

```
1  public class DrawFigures3 {
2      public static void main(String[] args) {
3          drawDiamond();
4          drawX();
5          drawRocket();
6      }
7
8      public static void drawDiamond() {
9          drawCone();
10         drawV();
11         System.out.println();
12     }
13
14     public static void drawX() {
15         drawV();
16         drawCone();
17         System.out.println();
18     }
19
20     public static void drawRocket() {
21         drawCone();
22         drawBox();
23         System.out.println("|United|");
24         System.out.println("|States|");
25         drawBox();
26         drawCone();
27         System.out.println();
28     }
29 }
```

```

30     public static void drawBox() {
31         System.out.println("+-----+");
32         System.out.println("|         |");
33         System.out.println("|         |");
34         System.out.println("+-----+");
35     }
36
37     public static void drawCone() {
38         System.out.println("  /\");
39         System.out.println(" /  \");
40         System.out.println("/    \");
41     }
42
43     public static void drawV() {
44         System.out.println("  \  /");
45         System.out.println("  \ /");
46         System.out.println("   \/");
47     }
48 }

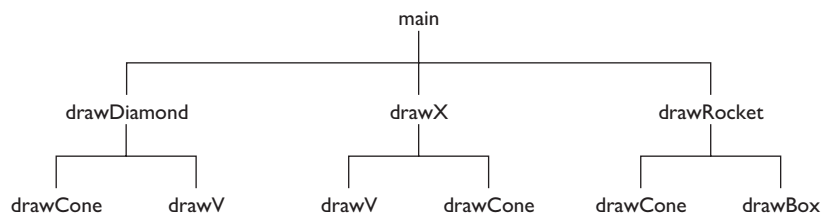
```

This program, now called `DrawFigures3`, has seven static methods defined within it. The first static method is the usual `main` method, which calls three methods. These three methods in turn call three other methods, which appear next.

### Analysis of Flow of Execution

The structure diagram in Figure 1.4 shows which static methods `main` calls and which static methods each of them calls. As you can see, this program has three levels of structure and two levels of decomposition. The overall task is split into three subtasks, each of which has two subtasks.

A program with methods has a more complex flow of control than one without them, but the rules are still fairly simple. Remember that when a method is called, the computer executes the statements in the body of that method. Then the computer proceeds to the next statement after the method call. Also remember that the computer always starts with the `main` method, executing its statements from first to last.



**Figure 1.4** Decomposition of `DrawFigures3`

So, to execute the `DrawFigures3` program, the computer first executes its `main` method. That, in turn, first executes the body of the method `drawDiamond`. `drawDiamond` executes the methods `drawCone` and `drawV` (in that order). When `drawDiamond` finishes executing, control shifts to the next statement in the body of the `main` method: the call to the `drawX` method.

A complete breakdown of the flow of control from static method to static method in `DrawFigures3` follows:

```
1st  main
2nd   drawDiamond
3rd    drawCone
4th    drawV
5th   drawX
6th    drawV
7th    drawCone
8th   drawRocket
9th    drawCone
10th   drawBox
11th   drawBox
12th   drawCone
```

Recall that the order in which you define methods does not have to parallel the order in which they are executed. The order of execution is determined by the body of the `main` method and by the bodies of methods called from `main`. A static method declaration is like a dictionary entry—it defines a word, but it does not specify how the word will be used. The body of this program's `main` method says to first execute `drawDiamond`, then `drawX`, then `drawRocket`. This is the order of execution, regardless of the order in which the methods are defined.

Java allows you to define methods in any order you like. Starting with `main` at the top and working down to lower and lower-level methods is a popular approach to take, but many people prefer the opposite, placing the low-level methods first and `main` at the end. Java doesn't care what order you use, so you can decide for yourself and do what you think is best. Consistency is important, though, so that you can easily find a method later in a large program.

It is important to note that the programs `DrawFigures1`, `DrawFigures2`, and `DrawFigures3` produce exactly the same output to the console. While `DrawFigures1` may be the easiest program for a novice to read, `DrawFigures2` and particularly `DrawFigures3` have many advantages over it. For one, a well-structured solution is easier to comprehend, and the methods themselves become a means of explaining the program. Also, programs with methods are more flexible and can more easily be adapted to similar but different tasks. You can take the seven methods defined in `DrawFigures3` and write a new program to produce a larger and more complex output. Building static methods to create new commands increases your flexibility without adding unnecessary complication. For example, you could replace the `main`

method with a version that calls the other methods in the following new order. What output would it produce?

```
public static void main(String[] args) {  
    drawCone();  
    drawCone();  
    drawRocket();  
    drawX();  
    drawRocket();  
    drawDiamond();  
    drawBox();  
    drawDiamond();  
    drawX();  
    drawRocket();  
}
```

## Chapter Summary

Computers execute sets of instructions called programs. Computers store information internally as sequences of 0s and 1s (binary numbers).

Programming and computer science deal with algorithms, which are step-by-step descriptions for solving problems.

Java is a modern object-oriented programming language developed by Sun Microsystems, now owned by Oracle Corporation, that has a large set of libraries you can use to build complex programs.

A program is translated from text into computer instructions by another program called a compiler. Java's compiler turns Java programs into a special format called Java bytecodes, which are executed using a special program called the Java Runtime Environment.

Java programmers typically complete their work using an editor called an Integrated Development Environment (IDE). The commands may vary from environment to

environment, but the same three-step process is always involved:

1. Type in a program as a Java class.
2. Compile the program file.
3. Run the compiled version of the program.

Java uses a command called `System.out.println` to display text on the console screen.

Written words in a program can take different meanings. Keywords are special reserved words that are part of the language. Identifiers are words defined by the programmer to name entities in the program. Words can also be put into strings, which are pieces of text that can be printed to the console.

Java programs that use proper spacing and layout are more readable to programmers. Readability is also improved by writing notes called comments inside the program.

The Java language has a syntax, or a legal set of commands that can be used. A Java program that does not follow the proper syntax will not compile. A program that does compile but that is written incorrectly may still contain errors called exceptions that occur when the program runs. A third kind of error is a logic or intent error. This kind of error occurs when the program runs but does not do what the programmer intended.

Commands in programs are called statements. A class can group statements into larger commands called static methods. Static methods help the programmer group code into

reusable pieces. An important static method that must be part of every program is called `main`.

Iterative enhancement is the process of building a program piece by piece, testing the program at each step before advancing to the next.

Complex programming tasks should be broken down into the major tasks the computer must perform. This process is called procedural decomposition. Correct use of static methods aids procedural decomposition.

## Self-Check Problems

### Section 1.1: Basic Computing Concepts

1. Why do computers use binary numbers?
2. Convert each of the following decimal numbers into its equivalent binary number:
  - a. 6
  - b. 44
  - c. 72
  - d. 131
3. What is the decimal equivalent of each of the following binary numbers?
  - a. 100
  - b. 1011
  - c. 101010
  - d. 1001110
4. In your own words, describe an algorithm for baking cookies. Assume that you have a large number of hungry friends, so you'll want to produce several batches of cookies!
5. What is the difference between the file `MyProgram.java` and the file `MyProgram.class`?

### Section 1.2: And Now—Java

6. Which of the following can be used in a Java program as identifiers?

<code>println</code>	<code>first-name</code>	<code>AnnualSalary</code>	<code>"hello"</code>	<code>ABC</code>
<code>42isTheAnswer</code>	<code>for</code>	<code>sum_of_data</code>	<code>_average</code>	<code>B4</code>

7. Which of the following is the correct syntax to output a message?

- a. `System.println(Hello, world!);`
- b. `System.println.out('Hello, world!');`
- c. `System.println("Hello, world!");`



```
d. System.out.println("Hello, world!");
e. Out.system.println("Hello, world!");
```

8. What is the output produced from the following statements?

```
System.out.println("\"Quotes\"");
System.out.println("Slashes \\/");
System.out.println("How '\"confounding' '\\\\' it is!");
```

9. What is the output produced from the following statements?

```
System.out.println("name\tage\theight");
System.out.println("Archie\t17\t5'9\"");
System.out.println("Betty\t17\t5'6\"");
System.out.println("Jughead\t16\t6'");
```

10. What is the output produced from the following statements?

```
System.out.println("Shaq is 7'1");
System.out.println("The string \"\" is an empty message.");
System.out.println("\\'\"");
```

11. What is the output produced from the following statements?

```
System.out.println("\ta\tb\tc");
System.out.println("\\\\");
System.out.println("");
System.out.println("\"\"");
System.out.println("C:\nin\the downward spiral");
```

12. What is the output produced from the following statements?

```
System.out.println("Dear \"DoubleSlash\" magazine,");
System.out.println();
System.out.println("\tYour publication confuses me. Is it");
System.out.println("a \\\\ slash or a \\\\ slash?");
System.out.println("\nSincerely,");
System.out.println("Susan \"Suzy\" Smith");
```

13. What series of println statements would produce the following output?

```
"Several slashes are sometimes seen,"
said Sally. "I've said so." See?
\ / \ \ // \ \ \ //
```

14. What series of println statements would produce the following output?

```
This is a test of your
knowledge of "quotes" used
in 'string literals.'

You're bound to "get it right"
if you read the section on
''quotes.''
```

15. Write a `println` statement that produces the following output:

```
/ \ // \ \ /// \ \ \
```

16. Rewrite the following code as a series of equivalent `System.out.println` statements (i.e., without any `System.out.print` statements):

```
System.out.print("Twas ");
System.out.print("brillig and the");
System.out.println(" ");
System.out.print(" slithy toves did");
System.out.print(" ");
System.out.println("gyre and");
System.out.println("gimble");
System.out.println();
System.out.println("in the wabe.");
```

17. What is the output of the following program? Note that the program contains several comments.

```
1  public class Commentary {
2      public static void main(String[] args) {
3          System.out.println("some lines of code");
4          System.out.println("have // characters on them");
5          System.out.println("which means "); // that they are comments
6          // System.out.println("written by the programmer.");
7
8          System.out.println("lines can also");
9          System.out.println("have /* and */ characters");
10         /* System.out.println("which represents");
11         System.out.println("a multi-line style");
12         */ System.out.println("of comment.");
13     }
14 }
```

### Section 1.3: Program Errors

18. Name the three errors in the following program:

```
1  public MyProgram {
2      public static void main(String[] args) {
3          System.out.println("This is a test of the")
4          System.out.Println("emergency broadcast system.");
5      }
6  }
```

19. Name the four errors in the following program:

```
1  public class SecretMessage {
2      public static main(string[] args) {
3          System.out.println("Speak friend");
```

```
4      System.out.println("and enter");
5
6  }
```

20. Name the four errors in the following program:

```
1  public class FamousSpeech
2      public static void main(String[]) {
3      System.out.println("Four score and seven years ago,");
4      System.out.println("our fathers brought forth on");
5      System.out.println("this continent a new nation");
6      System.out.println("conceived in liberty,");
7      System.out.println("and dedicated to the proposition");
8      System.out.println("that");      /* this part should
9      System.out.println("all");      really say,
10     System.out.println("men");      "all PEOPLE!" */
11     System.out.println("are";
12     System.out.println("created");
13     System.out.println("equal");
14     }
15 }
```

#### Section I.4: Procedural Decomposition

21. Which of the following method headers uses the correct syntax?

- a. public static example() {
- b. public static void example() {
- c. public void static example() {
- d. public static example void[] {
- e. public void static example{} (

22. What is the output of the following program? (You may wish to draw a structure diagram first.)

```
1  public class Tricky {
2      public static void main(String[] args) {
3          message1();
4          message2();
5          System.out.println("Done with main.");
6      }
7
8      public static void message1() {
9          System.out.println("This is message1.");
10     }
11
12     public static void message2() {
13         System.out.println("This is message2.");
14         message1();
15     }
16 }
```

```
15         System.out.println("Done with message2.");
16     }
17 }
```

23. What is the output of the following program? (You may wish to draw a structure diagram first.)

```
1  public class Strange {
2      public static void first() {
3          System.out.println("Inside first method");
4      }
5
6      public static void second() {
7          System.out.println("Inside second method");
8          first();
9      }
10
11     public static void third() {
12         System.out.println("Inside third method");
13         first();
14         second();
15     }
16
17     public static void main(String[] args) {
18         first();
19         third();
20         second();
21         third();
22     }
23 }
```

24. What would have been the output of the preceding program if the third method had contained the following statements?

```
public static void third() {
    first();
    second();
    System.out.println("Inside third method");
}
```

25. What would have been the output of the Strange program if the main method had contained the following statements? (Use the original version of third, not the modified version from the most recent exercise.)

```
public static void main(String[] args) {
    second();
    first();
    second();
    third();
}
```

26. What is the output of the following program? (You may wish to draw a structure diagram first.)

```

1  public class Confusing {
2      public static void method2() {
3          method1();
4          System.out.println("I am method 2.");
5      }
6
7      public static void method3() {
8          method2();
9          System.out.println("I am method 3.");
10         method1();
11     }
12
13     public static void method1() {
14         System.out.println("I am method 1.");
15     }
16
17     public static void main(String[] args) {
18         method1();
19         method3();
20         method2();
21         method3();
22     }
23 }
```

27. What would have been the output of the preceding program if the `method3` method had contained the following statements?

```

public static void method3() {
    method1();
    method2();
    System.out.println("I am method 3.");
}
```

28. What would have been the output of the `Confusing` program if the `main` method had contained the following statements? (Use the original version of `method3`, not the modified version from the most recent exercise.)

```

public static void main(String[] args) {
    method2();
    method1();
    method3();
    method2();
}
```

29. The following program contains at least 10 syntax errors. What are they?

```

1  public class LotsOf Errors {
2      public static main(String args) {
3          System.println(Hello, world!);
```

```

4         message()
5     }
6
7     public static void message {
8         System.out.println("This program surely cannot ";
9         System.out.println("have any "errors" in it");
10    }

```

30. Consider the following program, saved into a file named `Example.java`:

```

1  public class Example {
2      public static void displayRule() {
3          System.out.println("The first rule ");
4          System.out.println("of Java Club is,");
5          System.out.println();
6          System.out.println("you do not talk about Java Club.");
7      }
8
9      public static void main(String[] args) {
10         System.out.println("The rules of Java Club.");
11         displayRule();
12         displayRule();
13     }
14 }

```

What would happen if each of the following changes were made to the `Example` program? For example, would there be no effect, a syntax error, or a different program output? Treat each change independently of the others.

- Change line 1 to: `public class Demonstration`
- Change line 9 to: `public static void MAIN(String[] args) {`
- Insert a new line after line 11 that reads: `System.out.println();`
- Change line 2 to: `public static void printMessage() {`
- Change line 2 to: `public static void showMessage() {` and change lines 11 and 12 to: `showMessage();`
- Replace lines 3–4 with: `System.out.println("The first rule of Java Club is,");`

31. The following program is legal under Java's syntax rules, but it is difficult to read because of its layout and lack of comments. Reformat it using the rules given in this chapter, and add a comment header at the top of the program.

```

1  public
2  class GiveAdvice{ public static
3  void main (String[]args){ System.out.println (
4
5  "Programs can be easy or"); System.out.println(
6  "difficult to read, depending"
7  ); System.out.println("upon their format.")
8          ;System.out.println();System.out.println(
9  "Everyone, including yourself,");
10 System.out.println
11 ("will be happier if you choose");

```

```

12         System.out.println("to format your programs."
13     );    }
14     }

```

32. The following program is legal under Java's syntax rules, but it is difficult to read because of its layout and lack of comments. Reformat it using the rules given in this chapter, and add a comment header at the top of the program.

```

1  public
2  class Messy{public
3  static void main(String[] args){message ()
4      ;System.out.println()      ; message ( );}   public static void
5  message() { System.out.println(
6      "I really wish that"
7      );System.out.println
8  ("I had formatted my source")
9      ;System.out.println("code correctly!");}}

```

## Exercises

1. Write a complete Java program called `Stewie` that prints the following output:

```

////////////////////
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\

```

2. Write a complete Java program called `Spikey` that prints the following output:

```

\//
\\//
\\\\//
///\\
//\\
/\

```

3. Write a complete Java program called `wellFormed` that prints the following output:

```

A well-formed Java program has
a main method with { and }
braces.

```

```

A System.out.println statement
has ( and ) and usually a
String that starts and ends
with a " character.
(But we type \" instead!)

```

4. Write a complete Java program called `Difference` that prints the following output:

```

What is the difference between
a ' and a "? Or between a " and a \"?

```

```

One is what we see when we're typing our program.
The other is what appears on the "console."

```

5. Write a complete Java program called `MuchBetter` that prints the following output:

```
A "quoted" String is
'much' better if you learn
the rules of "escape sequences."
Also, "" represents an empty String.
Don't forget: use \" instead of " !
' ' is not the same as "
```

6. Write a complete Java program called `Meta` whose output is the text that would be the source code of a Java program that prints “Hello, world!” as its output.

7. Write a complete Java program called `Mantra` that prints the following output. Use at least one static method besides `main`.

There's one thing every coder must understand:  
The `System.out.println` command.

There's one thing every coder must understand:  
The `System.out.println` command.

8. Write a complete Java program called `Stewie2` that prints the following output. Use at least one static method besides `main`.

```

/////////////////////////////////
|| Victory is mine! ||
/////////////////////////////////
|| Victory is mine! ||
/////////////////////////////////
|| Victory is mine! ||
/////////////////////////////////
|| Victory is mine! ||
/////////////////////////////////
|| Victory is mine! ||
/////////////////////////////////

```

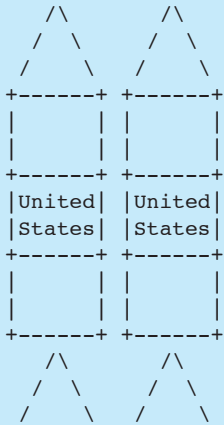
- 9. Write a program called Egg that displays the following output:**

10. Modify the program from the previous exercise to become a new program `Egg2` that displays the following output. Use static methods as appropriate.





11. Write a Java program called `TwoRockets` that generates the following output. Use static methods to show structure and eliminate redundancy in your solution. Note that there are two rocket ships next to each other. What redundancy can you eliminate using static methods? What redundancy cannot be eliminated?



12. Write a program called `FightSong` that produces this output. Use at least two static methods to show structure and eliminate redundancy in your solution.

```

Go, team, go!
You can do it.

Go, team, go!
You can do it.
You're the best,
In the West.
Go, team, go!
You can do it.

Go, team, go!
You can do it.
You're the best,
in the West.
Go, team, go!
You can do it.

Go, team, go!
You can do it.
```

13. Write a Java program called `StarFigures` that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.

```
*****
*****
 *  *
  *
 *  *
```

```
*****
*****
 *  *
  *
 *  *
*****
*****
```

```
  *
  *
  *
*****
*****
 *  *
  *
 *  *
```

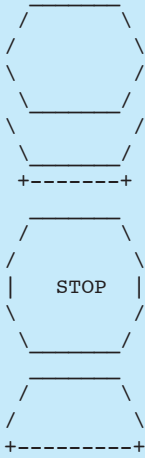
14. Write a Java program called `Lanterns` that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.

```
*****
*****
*****
```

```
*****
*****
*****
* | | | | *
*****
*****
*****
*****
```

```
*****
*****
*****
* | | | | *
* | | | | *
*****
*****
```

15. Write a Java program called `EggStop` that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.



16. Write a program called `Shining` that prints the following line of output 1000 times:

All work and no play makes Jack a dull boy.

You should not write a program that uses 1000 lines of source code; use methods to shorten the program. What is the shortest program you can write that will produce the 1000 lines of output, using only the material from this chapter?

## Programming Projects

1. Write a program to spell out MISSISSIPPI using block letters like the following (one per line):

```

M      M      I I I I      S S S S      P P P P P
M M    M M      I      S      S      P      P
M M M M      I      S      P      P
M  M  M      I      S S S S      P P P P P
M      M      I      S      P
M      M      I      S      S      P
M      M      I I I I      S S S S      P

```

2. Sometimes we write similar letters to different people. For example, you might write to your parents to tell them about your classes and your friends and to ask for money; you might write to a friend about your love life, your classes, and your hobbies; and you might write to your brother about your hobbies and your friends and to ask for money. Write a program that prints similar letters such as these to three people of your choice. Each letter should have at least one paragraph in common with each of the other letters. Your main program should have three method calls, one for each of the people to whom you are writing. Try to isolate repeated tasks into methods.
3. Write a program that produces as output the following lyrics. Use methods for each verse and the refrain. Here are the complete lyrics to print:

```

There was an old lady who swallowed a fly.
I don't know why she swallowed that fly,
Perhaps she'll die.

```

There was an old lady who swallowed a spider,  
That wriggled and iggled and jiggled inside her.  
She swallowed the spider to catch the fly,  
I don't know why she swallowed that fly,  
Perhaps she'll die.

There was an old lady who swallowed a bird,  
How absurd to swallow a bird.  
She swallowed the bird to catch the spider,  
She swallowed the spider to catch the fly,  
I don't know why she swallowed that fly,  
Perhaps she'll die.

There was an old lady who swallowed a cat,  
Imagine that to swallow a cat.  
She swallowed the cat to catch the bird,  
She swallowed the bird to catch the spider,  
She swallowed the spider to catch the fly,  
I don't know why she swallowed that fly,  
Perhaps she'll die.

There was an old lady who swallowed a dog,  
What a hog to swallow a dog.  
She swallowed the dog to catch the cat,  
She swallowed the cat to catch the bird,  
She swallowed the bird to catch the spider,  
She swallowed the spider to catch the fly,  
I don't know why she swallowed that fly,  
Perhaps she'll die.

There was an old lady who swallowed a horse,  
She died of course.

4. Write a program that produces as output the following lyrics. Use methods for each verse and the refrain. Here are the complete lyrics to print:

On the first day of Christmas,  
my true love sent to me  
a partridge in a pear tree.

On the second day of Christmas,  
my true love sent to me  
two turtle doves, and  
a partridge in a pear tree.

...

On the twelfth day of Christmas,  
my true love sent to me  
Twelve drummers drumming,  
eleven pipers piping,  
ten lords a-leaping,  
nine ladies dancing,  
eight maids a-milking,  
seven swans a-swimming,  
six geese a-laying,

```
five golden rings,  
four calling birds,  
three French hens,  
two turtle doves, and  
a partridge in a pear tree.
```

5. Write a program that produces as output the words of “The House That Jack Built.” Use methods for each verse and for repeated text. Here are lyrics to use:

```
This is the house that Jack built.
```

```
This is the malt  
That lay in the house that Jack built.
```

```
This is the rat,  
That ate the malt  
That lay in the house that Jack built.
```

```
This is the cat,  
That killed the rat,  
That ate the malt  
That lay in the house that Jack built.
```

```
This is the dog,  
That worried the cat,  
That killed the rat,  
That ate the malt  
That lay in the house that Jack built.
```

```
This is the cow with the crumpled horn,  
That tossed the dog,  
That worried the cat,  
That killed the rat,  
That ate the malt  
That lay in the house that Jack built.
```

```
This is the maiden all forlorn  
That milked the cow with the crumpled horn,  
That tossed the dog,  
That worried the cat,  
That killed the rat,  
That ate the malt  
That lay in the house that Jack built.
```

6. Write a program that produces as output the words of “Bought Me a Cat.” Use methods for each verse and for repeated text. Here are the song’s complete lyrics:

```
Bought me a cat and the cat pleased me,  
I fed my cat under yonder tree.  
Cat goes fiddle-i-fee.
```

```
Bought me a hen and the hen pleased me,  
I fed my hen under yonder tree.
```

Hen goes chimmy-chuck, chimmy-chuck,  
Cat goes fiddle-i-fee.

Bought me a duck and the duck pleased me,  
I fed my duck under yonder tree.  
Duck goes quack, quack,  
Hen goes chimmy-chuck, chimmy-chuck,  
Cat goes fiddle-i-fee.

Bought me a goose and the goose pleased me,  
I fed my goose under yonder tree.  
Goose goes hissy, hissy,  
Duck goes quack, quack,  
Hen goes chimmy-chuck, chimmy-chuck,  
Cat goes fiddle-i-fee.

Bought me a sheep and the sheep pleased me,  
I fed my sheep under yonder tree.  
Sheep goes baa, baa,  
Goose goes hissy, hissy,  
Duck goes quack, quack,  
Hen goes chimmy-chuck, chimmy-chuck,  
Cat goes fiddle-i-fee.

7. Write a program that produces as output the words of the following silly song. Use methods for each verse and for repeated text. Here are the song's complete lyrics:

I once wrote a program that wouldn't compile  
I don't know why it wouldn't compile,  
My TA just smiled.

My program did nothing  
So I started typing.  
I added `System.out.println("I <3 coding")`,  
I don't know why it wouldn't compile,  
My TA just smiled.

"Parse error," cried the compiler  
Luckily I'm such a code baller.  
I added a backslash to escape the quotes,  
I added `System.out.println("I <3 coding")`,  
I don't know why it wouldn't compile,  
My TA just smiled.

Now the compiler wanted an identifier  
And I thought the situation was getting dire.  
I added a main method with its `String[] args`,  
I added a backslash to escape the quotes,  
I added `System.out.println("I <3 coding")`,  
I don't know why it wouldn't compile,  
My TA just smiled.

```
Java complained it expected an enum
Boy, these computers really are dumb!
I added a public class and called it Scum,
I added a main method with its String[] args,
I added a backslash to escape the quotes,
I added System.out.println("I <3 coding"),
I don't know why it wouldn't compile,
My TA just smiled.
```

# Primitive Data and Definite Loops

## Introduction

Now that you know something about the basic structure of Java programs, you are ready to learn how to solve more complex problems. For the time being we will still concentrate on programs that produce output, but we will begin to explore some of the aspects of programming that require problem-solving skills.

The first half of this chapter fills in two important areas. First, it examines expressions, which are used to perform simple computations in Java, particularly those involving numeric data. Second, it discusses program elements called variables that can change in value as the program executes.

The second half of the chapter introduces your first control structure: the `for` loop. You use this structure to repeat actions in a program. This is useful whenever you find a pattern in a task such as the creation of a complex figure, because you can use a `for` loop to repeat the action to create that particular pattern. The challenge is finding each pattern and figuring out what repeated actions will reproduce it.

The `for` loop is a flexible control structure that can be used for many tasks. In this chapter we use it for *definite loops*, where you know exactly how many times you want to perform a particular task. In Chapter 5 we will discuss how to write *indefinite loops*, where you don't know in advance how many times to perform a task.

### 2.1 Basic Data Concepts

- Primitive Types
- Expressions
- Literals
- Arithmetic Operators
- Precedence
- Mixing Types and Casting

### 2.2 Variables

- Assignment/Declaration Variations
- String Concatenation
- Increment/Decrement Operators
- Variables and Mixing Types

### 2.3 The `for` Loop

- Tracing `for` Loops
- `for` Loop Patterns
- Nested `for` Loops

### 2.4 Managing Complexity

- Scope
- Pseudocode
- Class Constants

### 2.5 Case Study: Hourglass Figure

- Problem Decomposition and Pseudocode
- Initial Structured Version
- Adding a Class Constant
- Further Variations



2.1 Basic Data Concepts

Programs manipulate information, and information comes in many forms. Java is a *type-safe* language, which means that it requires you to be explicit about what kind of information you intend to manipulate and it guarantees that you manipulate the data in a reasonable manner. Everything that you manipulate in a Java program will be of a certain *type*, and you will constantly find yourself telling Java what types of data you intend to use.

Data Type

A name for a category of data values that are all related, as in type `int` in Java, which is used to represent integer values.

A decision was made early in the design of Java to support two different kinds of data: primitive data and objects. The designers made this decision purely on the basis of performance, to make Java programs run faster. Unfortunately, it means that you have to learn two sets of rules about how data works, but this is one of those times when you simply have to pay the price if you want to use an industrial-strength programming language. To make things a little easier, we will study the primitive data types first, in this chapter; in the next chapter, we will turn our attention to objects.

Primitive Types

There are eight primitive data types in Java: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`. Four of these are considered fundamental: `boolean`, `char`, `double`, and `int`. The other four types are variations that exist for programs that have special requirements. The four fundamental types that we will explore are listed in Table 2.1.

The type names (`int`, `double`, `char`, and `boolean`) are Java keywords that you will use in your programs to let the compiler know that you intend to use that type of data.

It may seem odd to use one type for integers and another type for real numbers. Isn't every integer a real number? The answer is yes, but these are fundamentally different types of numbers. The difference is so great that we make this distinction even in English. We don't ask, "How much sisters do you have?" or "How many do you weigh?" We realize that sisters come in discrete integer quantities (0 sisters, 1 sister, 2 sisters, 3 sisters, and so on), and we use the word "many" for integer quantities ("How

Table 2.1 Commonly Used Primitive Types in Java

Type	Description	Examples
<code>int</code>	integers (whole numbers)	42, -3, 18, 20493, 0
<code>double</code>	real numbers	7.35, 14.9, -19.83423
<code>char</code>	single characters	'a', 'X', '!'
<code>boolean</code>	logical values	true, false

many sisters do you have?”). Similarly, we realize that weight can vary by tiny amounts (175 pounds versus 175.5 pounds versus 175.25 pounds, and so on), and we use the word “much” for these real-number quantities (“How much do you weigh?”).

In programming, this distinction is even more important, because integers and reals are represented in different ways in the computer’s memory: Integers are stored exactly, while reals are stored as approximations with a limited number of digits of accuracy. You will see that storing values as approximations can lead to round-off errors when you use real values.

The name `double` for real values is not very intuitive. It’s an accident of history in much the same way that we still talk about “dialing” a number on our telephones even though modern telephones don’t have dials. The C programming language introduced a type called `float` (short for “floating-point number”) for storing real numbers. But `floats` had limited accuracy, so another type was introduced, called `double` (short for “double precision,” meaning that it had double the precision of a simple `float`). As memory became cheaper, people began using `double` as the default for floating-point values. In hindsight, it might have been better to use the word `float` for what is now called `double` and a word like “half” for the values with less accuracy, but it’s tough to change habits that are so ingrained. So, programming languages will continue to use the word `double` for floating-point numbers, and people will still talk about “dialing” people on the phone even if they’ve never touched a telephone dial.

## Expressions



When you write programs, you will often need to include values and calculations. The technical term for these elements is *expressions*.

### Expression

A simple value or a set of operations that produces a value.

The simplest expression is a specific value, like 42 or 28.9. We call these “literal values,” or *literals*. More complex expressions involve combining simple values. Suppose, for example, that you want to know how many bottles of water you have. If you have two 6-packs, four 4-packs, and two individual bottles, you can compute the total number of bottles with the following expression:

$$(2 * 6) + (4 * 4) + 2$$

Notice that we use an asterisk to represent multiplication and that we use parentheses to group parts of the expression. The computer determines the value of an expression by *evaluating* it.

### Evaluation

The process of obtaining the value of an expression.

The value obtained when an expression is evaluated is called the *result*. Complex expressions are formed using *operators*.

### Operator

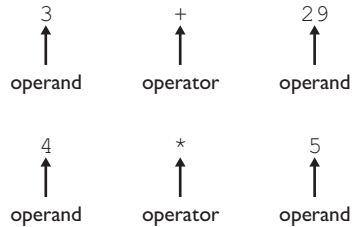
A special symbol (like + or \*) that is used to indicate an operation to be performed on one or more values.

The values used in the expression are called *operands*. For example, consider the following simple expressions:

$3 + 29$

$4 * 5$

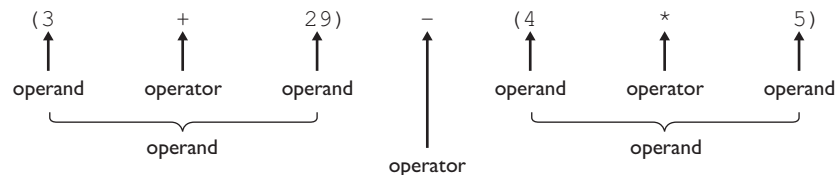
The operators here are the + and \*, and the operands are simple numbers.



When you form complex expressions, these simpler expressions can in turn become operands for other operators. For example, the expression

$(3 + 29) - (4 * 5)$

has two levels of operators.



The addition operator has simple operands of 3 and 29 and the multiplication operator has simple operands of 4 and 5, but the subtraction operator has operands that are each parenthesized expressions with operators of their own. Thus, complex expressions can be built from smaller expressions. At the lowest level, you have simple numbers. These are used as operands to make more complex expressions, which in turn can be used as operands in even more complex expressions.

There are many things you can do with expressions. One of the simplest things you can do is to print the value of an expression using a `println` statement. For example, if you say:

```
System.out.println(42);  
System.out.println(2 + 2);
```

you will get the following two lines of output:

```
42  
4
```

Notice that for the second `println`, the computer evaluates the expression (adding 2 and 2) and prints the result (in this case, 4).

You will see many different operators as you progress through this book, all of which can be used to form expressions. Expressions can be arbitrarily complex, with as many operators as you like. For that reason, when we tell you, “An expression can be used here,” we mean that you can use arbitrary expressions that include complex expressions as well as simple values.

## Literals

The simplest expressions refer to values directly using what are known as *literals*. An integer literal (considered to be of type `int`) is a sequence of digits with or without a leading sign:

```
3      482      -29434      0      92348      +9812
```

A floating-point literal (considered to be of type `double`) includes a decimal point:

```
298.4      0.284      207.      .2843      -17.452      -.98
```

Notice that `207.` is considered a `double` even though it coincides with an integer, because of the decimal point. Literals of type `double` can also be expressed in scientific notation (a number followed by `e` followed by an integer):

```
2.3e4      1e-5      3.84e92      2.458e12
```

The first of these numbers represents 2.3 times 10 to the 4th power, which equals 23,000. Even though this value happens to coincide with an integer, it is considered to be of type `double` because it is expressed in scientific notation. The second number represents 1 times 10 to the -5th power, which is equal to 0.00001. The third number represents 3.84 times 10 to the 92nd power. The fourth number represents 2.458 times 10 to the 12th power.

We have seen that textual information can be stored in literal strings that store a sequence of characters. In later chapters we will explore how to process a string

character by character. Each such character is of type `char`. A character literal is enclosed in single quotation marks and includes just one character:

```
'a'    'm'    'x'    '!'    '3'    '\\'
```

All of these examples are of type `char`. Notice that the last example uses an escape sequence to represent the backslash character. You can even refer to the single quotation character using an escape sequence:

```
'\''
```

Finally, the primitive type `boolean` stores logical information. We won't be exploring the use of type `boolean` until we reach Chapter 4 and see how to introduce logical tests into our programs, but for completeness, we include the `boolean` literal values here. Logic deals with just two possibilities: `true` and `false`. These two Java keywords are the two literal values of type `boolean`:

```
true    false
```

## Arithmetic Operators

The basic arithmetic operators are shown in Table 2.2. The addition and subtraction operators will, of course, look familiar to you, as should the asterisk as a multiplication operator and the forward slash as a division operator. However, as you'll see, Java has two different division operations. The remainder or mod operation may be unfamiliar.

Division presents a problem when the operands are integers. When you divide 119 by 5, for example, you do not get an integer result. Therefore, the results of integer division are expressed as two different integers, a quotient and a remainder:

$$\frac{119}{5} = 23 \text{ (quotient) with } 4 \text{ (remainder)}$$

In terms of the arithmetic operators:

```
119 / 5 evaluates to 23
```

```
119 % 5 evaluates to 4
```

**Table 2.2** Arithmetic Operators in Java

Operator	Meaning	Example	Result
+	addition	2 + 2	4
−	subtraction	53 − 18	35
*	multiplication	3 * 8	24
/	division	4.8 / 2.0	2.4
%	remainder or mod	19 % 5	4

These two division operators should be familiar if you recall how long-division calculations are performed:

$$\begin{array}{r} 31 \\ 34 \overline{)1079} \\ \underline{102} \phantom{00} \\ 59 \phantom{00} \\ \underline{34} \phantom{00} \\ 25 \end{array}$$

Here, dividing 1079 by 34 yields 31 with a remainder of 25. Using arithmetic operators, the problem would be described like this:

`1079 / 34` evaluates to 31

`1079 % 34` evaluates to 25

It takes a while to get used to integer division in Java. When you are using the division operator (`/`), the key thing to keep in mind is that it truncates anything after the decimal point. So, if you imagine computing an answer on a calculator, just think of ignoring anything after the decimal point:

- `19/5` is 3.8 on a calculator, so `19/5` evaluates to 3
- `207/10` is 20.7 on a calculator, so `207/10` evaluates to 20
- `3/8` is 0.375 on a calculator, so `3/8` evaluates to 0

The remainder operator (`%`) is usually referred to as the “mod operator,” or simply “mod.” The mod operator lets you know how much was left unaccounted for by the truncating division operator. For example, given the previous examples, you’d compute the mod results as shown in Table 2.3.

In each case, you figure out how much of the number is accounted for by the truncating division operator. The mod operator gives you any excess (the remainder). When you put this into a formula, you can think of the mod operator as behaving as follows:

$$x \% y = x - (x / y) * y$$

**Table 2.3 Examples of Mod Operator**

Mod problem	First divide	What does division account for?	How much is left over?	Answer
<code>19 % 5</code>	<code>19/5</code> is 3	<code>3 * 5</code> is 15	<code>19 - 15</code> is 4	4
<code>207 % 10</code>	<code>207/10</code> is 20	<code>20 * 10</code> is 200	<code>207 - 200</code> is 7	7
<code>3 % 8</code>	<code>3/8</code> is 0	<code>0 * 8</code> is 0	<code>3 - 0</code> is 3	3

It is possible to get a result of 0 for the mod operator. This happens when one number divides evenly into another. For example, each of the following expressions evaluates to 0 because the second number goes evenly into the first number:

```
28 % 7
95 % 5
44 % 2
```

A few special cases are worth noting because they are not always immediately obvious to novice programmers:

- **Numerator smaller than denominator:** In this case division produces 0 and mod produces the original number. For example,  $7 / 10$  is 0 and  $7 \% 10$  is 7.
- **Numerator of 0:** In this case both division and mod return 0. For example, both  $0 / 10$  and  $0 \% 10$  evaluate to 0.
- **Denominator of 0:** In this case, both division and mod are undefined and produce a runtime error. For example, a program that attempts to evaluate either  $7 / 0$  or  $7 \% 0$  will throw an `ArithmeticException` error.

The mod operator has many useful applications in computer programs. Here are just a few ideas:

- Testing whether a number is even or odd ( $\text{number} \% 2$  is 0 for evens,  $\text{number} \% 2$  is 1 for odds).
- Finding individual digits of a number (e.g.,  $\text{number} \% 10$  is the final digit).
- Finding the last four digits of a social security number ( $\text{number} \% 10000$ ).

The remainder operator can be used with `doubles` as well as with integers, and it works similarly: You consider how much is left over when you take away as many “whole” values as you can. For example, the expression  $10.2 \% 2.4$  evaluates to 0.6 because you can take away four 2.4s from 10.2, leaving you with 0.6 left over.

For floating-point values (values of type `double`), the division operator does what we consider “normal” division. So, even though the expression  $119 / 5$  evaluates to 23, the expression  $119.0 / 5.0$  evaluates to 23.8.

## Precedence

Java expressions are like complex noun phrases in English. Such phrases are subject to ambiguity. For example, consider the phrase “the man on the hill by the river with the telescope.” Is the river by the hill or by the man? Is the man holding the telescope, or is the telescope on the hill, or is the telescope in the river? We don’t know how to group the various parts together.

You can get the same kind of ambiguity if parentheses aren’t used to group the parts of a Java expression. For example, the expression  $2 + 3 * 4$  has two operators. Which operation is performed first? You could interpret this two ways:

$$\begin{array}{ccccc}
 2 & + & 3 & * & 4 \\
 \underbrace{\hspace{1.5cm}} & & & & \\
 5 & & & * & 4 \\
 \underbrace{\hspace{2.5cm}} & & & & \\
 & & 20 & & 
 \end{array}
 \qquad
 \begin{array}{ccccc}
 2 & + & 3 & * & 4 \\
 & & \underbrace{\hspace{1.5cm}} & & \\
 & & 12 & & \\
 \underbrace{\hspace{1.5cm}} & + & & & \\
 2 & + & & & \\
 \underbrace{\hspace{2.5cm}} & & & & \\
 & & 14 & & 
 \end{array}$$

The first of these evaluates to 20 while the second evaluates to 14. To deal with the ambiguity, Java has rules of *precedence* that determine how to group together the various parts.

### Precedence

The binding power of an operator, which determines how to group parts of an expression.

The computer applies rules of precedence when the grouping of operators in an expression is ambiguous. An operator with high precedence is evaluated first, followed by operators of lower precedence. Within a given level of precedence the operators are evaluated in one direction, usually left to right.

For arithmetic expressions, there are two levels of precedence. The multiplicative operators (\*, /, %) have a higher level of precedence than the additive operators (+, -). Thus, the expression  $2 + 3 * 4$  is interpreted as

$$\begin{array}{ccccc}
 2 & + & 3 & * & 4 \\
 & & \underbrace{\hspace{1.5cm}} & & \\
 & & 12 & & \\
 \underbrace{\hspace{1.5cm}} & + & & & \\
 2 & + & & & \\
 \underbrace{\hspace{2.5cm}} & & & & \\
 & & 14 & & 
 \end{array}$$

Within the same level of precedence, arithmetic operators are evaluated from left to right. This often doesn't make a difference in the final result, but occasionally it does. Consider, for example, the expression

$$40 - 25 - 9$$

which evaluates as follows:

$$\begin{array}{ccccc}
 40 & - & 25 & - & 9 \\
 \underbrace{\hspace{1.5cm}} & & & & \\
 15 & & & - & 9 \\
 & & \underbrace{\hspace{1.5cm}} & & \\
 & & 6 & & 
 \end{array}$$

You would get a different result if the second subtraction were evaluated first.

You can always override precedence with parentheses. For example, if you really want the second subtraction to be evaluated first, you can force that to happen by introducing parentheses:

$$40 - (25 - 9)$$



Table 2.4 Java Operator Precedence

Description	Operators
unary operators	<code>+</code> , <code>-</code>
multiplicative operators	<code>*</code> , <code>/</code> , <code>%</code>
additive operators	<code>+</code> , <code>-</code>

The expression now evaluates as follows:

$$\begin{array}{rcl} 40 & - & (25 - 9) \\ & & \underbrace{\hspace{1.5cm}} \\ 40 & - & 16 \\ \underbrace{\hspace{1.5cm}} & & \\ 24 & & \end{array}$$

Another concept in arithmetic is *unary* plus and minus, which take a single operand, as opposed to the binary operators we have seen thus far (e.g., `*`, `/`, and even binary `+` and `-`), all of which take two operands. For example, we can find the negation of 8 by asking for `-8`. These unary operators have a higher level of precedence than the multiplicative operators. Consequently, we can form expressions like the following:

`12 * -8`

which evaluates to `-96`.

We will see many types of operators in the next few chapters. Table 2.4 is a precedence table that includes the arithmetic operators. As we introduce more operators, we'll update this table to include them as well. The table is ordered from highest precedence to lowest precedence and indicates that Java will first group parts of an expression using the unary operators, then using the multiplicative operators, and finally using the additive operators.

Before we leave this topic, let's look at a complex expression and see how it is evaluated step by step. Consider the following expression:

`13 * 2 + 239 / 10 % 5 - 2 * 2`

It has a total of six operators: two multiplications, one division, one mod, one subtraction, and one addition. The multiplication, division, and mod operations will be performed first, because they have higher precedence, and they will be performed from left to right because they are all at the same level of precedence:

$$\begin{array}{rcl} 13 & * & 2 & + & 239 & / & 10 & \% & 5 & - & 2 & * & 2 \\ \underbrace{\hspace{1.5cm}} & & & & & & & & & & & & \\ 26 & & & + & 239 & / & 10 & \% & 5 & - & 2 & * & 2 \\ & & & & & & \underbrace{\hspace{1.5cm}} & & & & & & \\ 26 & & + & & 23 & \% & 5 & - & 2 & * & 2 \\ & & & & & & \underbrace{\hspace{1.5cm}} & & & & & & \\ 26 & & + & & 3 & & & - & 2 & * & 2 \\ & & & & & & & & & \underbrace{\hspace{1.5cm}} & & \\ 26 & & + & & 3 & & & - & & 4 & & \end{array}$$

Now we evaluate the additive operators from left to right:

$$\begin{array}{ccccccc}
 26 & + & 3 & - & 4 \\
 & \underbrace{\hspace{1.5cm}} & & & \\
 & 29 & - & 4 \\
 & & \underbrace{\hspace{1.5cm}} & & \\
 & & 25 & & 
 \end{array}$$

## Mixing Types and Casting

You'll often find yourself mixing values of different types and wanting to convert from one type to another. Java has simple rules to avoid confusion and provides a mechanism for requesting that a value be converted from one type to another.

Two types that are frequently mixed are `ints` and `doubles`. You might, for example, ask Java to compute `2 * 3.6`. This expression includes the `int` literal `2` and the `double` literal `3.6`. In this case, Java converts the `int` into a `double` and performs the computation entirely with `double` values; this is always the rule when Java encounters an `int` where it was expecting a `double`.

This becomes particularly important when you form expressions that involve division. If the two operands are both of type `int`, Java will use integer (truncating) division. If either of the two operands is of type `double`, however, it will do real-valued (normal) division. For example, `23 / 4` evaluates to `5`, but all of the following evaluate to `5.75`:

```

23.0 / 4
23. / 4
23 / 4.0
23 / 4.
23. / 4.
23.0 / 4.0

```

Sometimes you want Java to go the other way, converting a `double` into an `int`. You can ask Java for this conversion with a *cast*. Think of it as “casting a value in a different light.” You request a cast by putting the name of the type you want to cast to in parentheses in front of the value you want to cast. For example,

```
(int) 4.75
```

will produce the `int` value `4`. When you cast a `double` value to an `int`, it simply truncates anything after the decimal point.

If you want to cast the result of an expression, you have to be careful to use parentheses. For example, suppose that you have some books that are each `0.15` feet wide and you want to know how many of them will fit in a bookshelf that is `2.5` feet wide. You could do a straight division of `2.5 / 0.15`, but that evaluates to a `double` result that is between `16` and `17`. Americans use the phrase “`16` and change” as a way to express the idea that a value is larger than `16` but not as big as `17`. In this case, we

don't care about the "change"; we only want to compute the 16 part. You might form the following expression:

```
(int) 2.5 / 0.15
```

Unfortunately, this expression evaluates to the wrong answer because the cast is applied to whatever comes right after it (here, the value 2.5). This casts 2.5 into the integer 2, divides by 0.15, and evaluates to 13 and change, which isn't an integer and isn't the right answer. Instead, you want to form this expression:

```
(int) (2.5 / 0.15)
```

This expression first performs the division to get 16 and change, and then casts that value to an `int` by truncating it. It thus evaluates to the `int` value 16, which is the answer you're looking for.

## 2.2 Variables



VideoNote

Primitive data can be stored in the computer's memory in a *variable*.

### Variable

A memory location with a name and a type that stores a value.

Think of the computer's memory as being like a giant spreadsheet that has many cells where data can be stored. When you create a variable in Java, you are asking it to set aside one of those cells for this new variable. Initially the cell will be empty, but you will have the option to store a value in the cell. And as with a spreadsheet, you will have the option to change the value in that cell later.

Java is a little more picky than a spreadsheet, though, in that it requires you to tell it exactly what kind of data you are going to store in the cell. For example, if you want to store an integer, you need to tell Java that you intend to use type `int`. If you want to store a real value, you need to tell Java that you intend to use a `double`. You also have to decide on a name to use when you want to refer to this memory location. The normal rules of Java identifiers apply (the name must start with a letter, which can be followed by any combination of letters and digits). The standard convention in Java is to start variable names with a lowercase letter, as in `number` or `digits`, and to capitalize any subsequent words, as in `numberOfDigits`.

To explore the basic use of variables, let's examine a program that computes an individual's *body mass index* (BMI). Health professionals use this number to advise people about whether or not they are overweight. Given an individual's height and weight, we can compute that person's BMI. A simple BMI program, then, would naturally have three variables for these three pieces of information. There are several details that we need to discuss about variables, but it can be helpful to look at a complete

program first to see the overall picture. The following program computes and prints the BMI for an individual who is 5 feet 10 inches tall and weighs 195 pounds:

```
1  public class BMICalculator {
2      public static void main(String[] args) {
3          // declare variables
4          double height;
5          double weight;
6          double bmi;
7
8          // compute BMI
9          height = 70;
10         weight = 195;
11         bmi = weight / (height * height) * 703;
12
13         // print results
14         System.out.println("Current BMI:");
15         System.out.println(bmi);
16     }
17 }
```

Notice that the program includes blank lines to separate the sections and comments to indicate what the different parts of the program do. It produces the following output:

```
Current BMI:
27.976530612244897
```

Let's now examine the details of this program to understand how variables work. Before variables can be used in a Java program, they must be declared. The line of code that declares the variable is known as a variable *declaration*.

### Declaration

A request to set aside a new variable with a given name and type.

Each variable is declared just once. If you declare a variable more than once, you will get an error message from the Java compiler. Simple variable declarations are of the form

```
<type> <name>;
```

as in the three declarations at the beginning of our sample program:

```
double height;
double weight;
double bmi;
```

Notice that a variable declaration, like a statement, ends with a semicolon. These declarations can appear anywhere a statement can occur. The declaration indicates the type and the name of the variable. Remember that the name of each primitive type is a keyword in Java (`int`, `double`, `char`, `boolean`). We've used the keyword `double` to define the type of these three variables.

Once a variable is declared, Java sets aside a memory location to store its value. However, with the simple form of variable declaration used in our program, Java does not store initial values in these memory locations. We refer to these as *uninitialized* variables, and they are similar to blank cells in a spreadsheet:

height ?      weight ?      bmi ?

So how do we get values into those cells? The easiest way to do so is using an *assignment statement*. The general syntax of the assignment statement is

```
<variable> = <expression>;
```

as in

```
height = 70;
```

This statement stores the value 70 in the memory location for the variable `height`, indicating that this person is 70 inches tall (5 feet 10 inches). We often use the phrase “gets” or “is assigned” when reading a statement like this, as in “`height` gets 70” or “`height` is assigned 70.”

When the statement executes, the computer first evaluates the expression on the right side; then, it stores the result in the memory location for the given variable. In this case the expression is just a simple literal value, so after the computer executes this statement, the memory looks like this:

height 70.0      weight ?      bmi ?

Notice that the value is stored as `70.0` because the variable is of type `double`. The variable `height` has now been initialized, but the variables `weight` and `bmi` are still uninitialized. The second assignment statement gives a value to `weight`:

```
weight = 195;
```

After executing this statement, the memory looks like this:

height 70.0      weight 195.0      bmi ?

The third assignment statement includes a formula (an expression to be evaluated):

```
bmi = weight / (height * height) * 703;
```

To calculate the value of this expression, the computer divides the weight by the square of the height and then multiplies the result of that operation by the literal value 703. The result is stored in the variable `bmi`. So, after the computer has executed the third assignment statement, the memory looks like this:

height	70.0	weight	195.0	bmi	27.976530612244897
--------	------	--------	-------	-----	--------------------

The last two lines of the program report the BMI result using `println` statements:

```
System.out.println("Current BMI:");  
System.out.println(bmi);
```

Notice that we can include a variable in a `println` statement the same way that we include literal values and other expressions to be printed.

As its name implies, a variable can take on different values at different times. For example, consider the following variation of the BMI program, which computes a new BMI assuming the person lost 15 pounds (going from 195 pounds to 180 pounds).

```
1 public class BMICalculator2 {  
2     public static void main(String[] args) {  
3         // declare variables  
4         double height;  
5         double weight;  
6         double bmi;  
7  
8         // compute BMI  
9         height = 70;  
10        weight = 195;  
11        bmi = weight / (height * height) * 703;  
12  
13        // print results  
14        System.out.println("Previous BMI:");  
15        System.out.println(bmi);  
16  
17        // recompute BMI  
18        weight = 180;  
19        bmi = weight / (height * height) * 703;  
20  
21        // report new results  
22        System.out.println("Current BMI:");  
23        System.out.println(bmi);  
24    }  
25 }
```

The program begins the same way, setting the three variables to the following values and reporting this initial value for BMI:

height 70.0      weight 195.0      bmi 27.976530612244897

But the new program then includes the following assignment statement:

```
weight = 180;
```

This changes the value of the `weight` variable:

height 70.0      weight 180.0      bmi 27.976530612244897

You might think that this would also change the value of the `bmi` variable. After all, earlier in the program we said that the following should be true:

```
bmi = weight / (height * height) * 703;
```

This is a place where the spreadsheet analogy is not as accurate. A spreadsheet can store formulas in its cells and when you update one cell it can cause the values in other cells to be updated. The same is not true in Java.

You might also be misled by the use of an equals sign for assignment. Don't confuse this statement with a statement of equality. The assignment statement does not represent an algebraic relationship. In algebra, you might say

$$x = y + 2$$

In mathematics you state definitively that  $x$  is equal to  $y$  plus two, a fact that is true now and forever. If  $x$  changes,  $y$  will change accordingly, and vice versa. Java's assignment statement is very different.

The assignment statement is a command to perform an action at a particular point in time. It does not represent a lasting relationship between variables. That's why we usually say "gets" or "is assigned" rather than saying "equals" when we read assignment statements.

Getting back to the program, resetting the variable called `weight` does not reset the variable called `bmi`. To recompute `bmi` based on the new value for `weight`, we must include the second assignment statement:

```
weight = 180;
bmi = weight / (height * height) * 703;
```

Otherwise, the variable `bmi` would store the same value as before. That would be a rather depressing outcome to report to someone who's just lost 15 pounds. By including both of these statements, we reset both the `weight` and `bmi` variables so that memory looks like this:

height 70.0      weight 180.0      bmi 25.82448979591837

The output of the new version of the program is

```
Previous BMI:
27.976530612244897
Current BMI:
25.82448979591837
```

One very common assignment statement that points out the difference between algebraic relationships and program statements is:

```
x = x + 1;
```

Remember not to think of this as “ $x$  equals  $x + 1$ .” There are no numbers that satisfy that equation. We use a word like “gets” to read this as “ $x$  gets the value of  $x$  plus one.” This may seem a rather odd statement, but you should be able to decipher it given the rules outlined earlier. Suppose that the current value of  $x$  is 19. To execute the statement, you first evaluate the expression to obtain the result 20. The computer stores this value in the variable named on the left,  $x$ . Thus, this statement adds one to the value of the variable. We refer to this as *incrementing* the value of  $x$ . It is a fundamental programming operation because it is the programming equivalent of counting (1, 2, 3, 4, and so on). The following statement is a variation that counts down, which we call *decrementing* a variable:

```
x = x - 1;
```

We will discuss incrementing and decrementing in more detail later in this chapter.

## Assignment/Declaration Variations

Java is a complex language that provides a lot of flexibility to programmers. In the last section we saw the simplest form of variable declaration and assignment, but there are many variations on this theme. It wouldn’t be a bad idea to stick with the simplest form while you are learning, but you’ll come across other forms as you read other people’s programs, so you’ll want to understand what they mean.

The first variation is that Java allows you to provide an initial value for a variable at the time that you declare it. The syntax is as follows:

```
<type> <name> = <expression>;
```

as in

```
double height = 70;
double weight = 195;
double bmi = weight / (height * height) * 703;
```

This variation combines declaration and assignment in one line of code. The first two assignments have simple numbers after the equals sign, but the third has a complex



expression after the equals sign. These three assignments have the same effect as providing three declarations followed by three assignment statements:

```
double height;  
double weight;  
double bmi;  
height = 70;  
weight = 195;  
bmi = weight / (height * height) * 703;
```

Another variation is to declare several variables that are all of the same type in a single statement. The syntax is

```
<type> <name>, <name>, <name>, ..., <name>;
```

as in

```
double height, weight;
```

This example declares two different variables, both of type `double`. Notice that the type appears just once, at the beginning of the declaration.

The final variation is a mixture of the previous two forms. You can declare multiple variables all of the same type, and you can initialize them at the same time. For example, you could say

```
double height = 70, weight = 195;
```

This statement declares the two `double` variables `height` and `weight` and gives them initial values (70 and 195, respectively). Java even allows you to mix initializing and not initializing, as in

```
double height = 70, weight = 195, bmi;
```

This statement declares three `double` variables called `height`, `weight`, and `bmi` and provides initial values to two of them (`height` and `weight`). The variable `bmi` is uninitialized.

### Common Programming Error

#### Declaring the Same Variable Twice

One of the things to keep in mind as you learn is that you can declare any given variable just once. You can assign it as many times as you like once you've declared it, but the declaration should appear just once. Think of variable declaration as being like checking into a hotel and assignment as being like going in

*Continued on next page*

*Continued from previous page*

and out of your room. You have to check in first to get your room key, but then you can come and go as often as you like. If you tried to check in a second time, the hotel would be likely to ask you if you really want to pay for a second room.

If you declare a variable more than once, Java generates a compiler error. For example, say your program contains the following lines:



```
int x = 13;
System.out.println(x);
int x = 2;           // this line does not compile
System.out.println(x);
```

The first line is okay. It declares an integer variable called `x` and initializes it to 13. The second line is also okay, because it simply prints the value of `x`. But the third line will generate an error message indicating that “`x` is already defined.” If you want to change the value of `x` you need to use a simple assignment statement instead of a variable declaration:

```
int x = 13;
System.out.println(x);
x = 2;
System.out.println(x);
```

We have been referring to the “assignment statement,” but in fact assignment is an operator, not a statement. When you assign a value to a variable, the overall expression evaluates to the value just assigned. That means that you can form expressions that have assignment operators embedded within them. Unlike most other operators, the assignment operator evaluates from right to left, which allows programmers to write statements like the following:

```
int x, y, z;
x = y = z = 2 * 5 + 4;
```

Because the assignment operator evaluates from right to left, this statement is equivalent to:

```
x = (y = (z = 2 * 5 + 4));
```

The expression `2 * 5 + 4` evaluates to 14. This value is assigned to `z`. The assignment is itself an expression that evaluates to 14, which is then assigned to `y`. The assignment to `y` evaluates to 14 as well, which is then assigned to `x`. The result is that all three variables are assigned the value 14.

## String Concatenation

You saw in Chapter 1 that you can output string literals using `System.out.println`. You can also output numeric expressions using `System.out.println`:

```
System.out.println(12 + 3 - 1);
```

This statement causes the computer first to evaluate the expression, which yields the value 14, and then to write that value to the console window. You'll often want to output more than one value on a line, but unfortunately, you can pass only one value to `println`. To get around this limitation, Java provides a simple mechanism called *concatenation* for putting together several pieces into one long string literal.

### String Concatenation

Combining several strings into a single string, or combining a string with other data into a new, longer string.

The addition (+) operator concatenates the pieces together. Doing so forms an expression that can be evaluated. Even if the expression includes both numbers and text, it can be evaluated just like the numeric expressions we have been exploring. Consider, for example, the following:

```
"I have " + 3 + " things to concatenate"
```

You have to pay close attention to the quotation marks in an expression like this to keep track of which parts are “inside” a string literal and which are outside. This expression begins with the text `"I have "` (including a space at the end), followed by a plus sign and the integer literal 3. Java converts the integer into a textual form (`"3"`) and concatenates the two pieces together to form `"I have 3"`. Following the 3 is another plus and another string literal, `"things to concatenate"` (which starts with a space). This piece is glued onto the end of the previous string to form the string `"I have 3 things to concatenate"`.

Because this expression produces a single concatenated string, we can include it in a `println` statement:

```
System.out.println("I have " + 3 + " things to concatenate");
```

This statement produces a single line of output:

```
I have 3 things to concatenate
```

String concatenation is often used to report the value of a variable. Consider, for example, the following program that computes the number of hours, minutes, and seconds in a standard year:

```
1 public class Time {  
2     public static void main(String[] args) {  
3         int hours = 365 * 24;
```

```

4         int minutes = hours * 60;
5         int seconds = minutes * 60;
6         System.out.println("Hours in a year = " + hours);
7         System.out.println("Minutes in a year = " + minutes);
8         System.out.println("Seconds in a year = " + seconds);
9     }
10 }

```

Notice that the three `println` commands at the end each have a string literal concatenated with a variable. The program produces the following output:

```

Hours in a year = 8760
Minutes in a year = 525600
Seconds in a year = 31536000

```

You can use concatenation to form arbitrarily complex expressions. For example, if you had variables `x`, `y`, and `z` and you wanted to write out their values in coordinate format with parentheses and commas, you could say:

```
System.out.println("(" + x + ", " + y + ", " + z + ")");
```

If `x`, `y`, and `z` had the values 8, 19, and 23, respectively, this statement would output the string `"(8, 19, 23)"`.

The `+` used for concatenation has the same level of precedence as the normal arithmetic `+` operator, which can lead to some confusion. Consider, for example, the following expression:

```
2 + 3 + " hello " + 7 + 2 * 3
```

This expression has four addition operators and one multiplication operator. Because of precedence, we evaluate the multiplication first:

```

2 + 3 + " hello " + 7 + 2 * 3
                        ⏟
2 + 3 + " hello " + 7 + 6

```

This grouping might seem odd, but that's what the precedence rule says to do: We don't evaluate any additive operators until we've first evaluated all of the multiplicative operators. Once we've taken care of the multiplication, we're left with the four addition operators. These will be evaluated from left to right.

The first addition involves two integer values. Even though the overall expression involves a string, because this little subexpression has just two integers we perform integer addition:

```

2 + 3 + " hello " + 7 + 6
  ⏟
  5 + " hello " + 7 + 6

```

The next addition involves adding the integer 5 to the string literal " hello ". If either of the two operands is a string, we perform concatenation. So, in this case, we convert the integer into a text equivalent ( "5" ) and glue the pieces together to form a new string value:

$$\begin{array}{ccccccc} 5 & + & \text{" hello "}& + & 7 & + & 6 \\ & & \underbrace{\hspace{1.5cm}} & & & & \\ & & \text{"5 hello "}& + & 7 & + & 6 \end{array}$$

You might think that Java would add together the 7 and 6 the same way it added the 2 and 3 to make 5. But it doesn't work that way. The rules of precedence are simple, and Java follows them with simple-minded consistency. Precedence tells us that addition operators are evaluated from left to right, so first we add the string "5 hello " to 7. That is another combination of a string and an integer, so Java converts the integer to its textual equivalent ( "7" ) and concatenates the two parts together to form a new string:

$$\begin{array}{ccccccc} \text{"5 hello "}& + & 7 & + & 6 \\ & & \underbrace{\hspace{1.5cm}} & & \\ & & \text{"5 hello 7"}& + & 6 \end{array}$$

Now there is just a single remaining addition to perform, which again involves a string/integer combination. We convert the integer to its textual equivalent ( "6" ) and concatenate the two parts together to form a new string:

$$\begin{array}{ccccccc} \text{"5 hello 7"}& + & 6 \\ & & \underbrace{\hspace{1.5cm}} & & \\ & & \text{"5 hello 76"}& & \end{array}$$

Clearly, such expressions can be confusing, but you wouldn't want the Java compiler to have to try to guess what you mean. Our job as programmers is easier if we know that the compiler is going to follow simple rules consistently. You can make the expression clearer, and specify how it is evaluated, by adding parentheses. For example, if we really did want Java to add together the 7 and 6 instead of concatenating them separately, we could have written the original expression in the following much clearer way:

```
(2 + 3) + " hello " + (7 + 2 * 3)
```

Because of the parentheses, Java will evaluate the two numeric parts of this expression first and then concatenate the results with the string in the middle. This expression evaluates to "5 hello 13".

## Increment/Decrement Operators

In addition to the standard assignment operator, Java has several special operators that are useful for a particular family of operations that are common in programming. As we mentioned earlier, you will often find yourself increasing the value of a variable by a particular amount, an operation called *incrementing*. You will also often

find yourself decreasing the value of a variable by a particular amount, an operation called *decrementing*. To accomplish this, you write statements like the following:

```
x = x + 1;  
y = y - 1;  
z = z + 2;
```

Likewise, you'll frequently find yourself wanting to double or triple the value of a variable or to reduce its value by a factor of 2, in which case you might write code like the following:

```
x = x * 2;  
y = y * 3;  
z = z / 2;
```

Java has a shorthand for these situations. You glue together the operator character (+, −, \*, etc.) with the equals sign to get a special assignment operator (+=, −=, \*=, etc.). This variation allows you to rewrite assignment statements like the previous ones as follows:

```
x += 1;  
y -= 1;  
z += 2;  
  
x *= 2;  
y *= 3;  
z /= 2;
```

This convention is yet another detail to learn about Java, but it can make the code easier to read. Think of a statement like `x += 2` as saying, “add 2 to x.” That’s more concise than saying `x = x + 2`.

Java has an even more concise way of expressing the particular case in which you want to increment by 1 or decrement by 1. In this case, you can use the increment and decrement operators (++ and --). For example, you can say

```
x++;  
y--;
```

There are actually two different forms of each of these operators, because you can also put the operator in front of the variable:

```
++x;  
--y;
```

The two versions of ++ are known as the preincrement (++x) and postincrement (x++) operators. The two versions of -- are similarly known as the predecrement

Table 2.5 Java Operator Precedence

Description	Operators
unary operators	++, --, +, -
multiplicative operators	*, /, %
additive operators	+, -
assignment operators	=, +=, -=, *=, /=, %=

(--x) and postdecrement (x--) operators. The pre- versus post- distinction doesn't matter when you include them as statements by themselves, as in these two examples. The difference comes up only when you embed these statements inside more complex expressions, which we don't recommend.

Now that we've seen a number of new operators, it is worth revisiting the issue of precedence. Table 2.5 shows an updated version of the Java operator precedence table that includes the assignment operators and the increment and decrement operators. Notice that the increment and decrement operators are grouped with the unary operators and have the highest precedence.

Did You Know?

++ and --

The ++ and -- operators were first introduced in the C programming language. Java has them because the designers of the language decided to use the syntax of C as the basis for Java syntax. Many languages have made the same choice, including C++ and C#. There is almost a sense of pride among C programmers that these operators allow you to write extremely concise code, but many other people feel that they can make code unnecessarily complex. In this book we always use these operators as separate statements so that it is obvious what is going on, but in the interest of completeness we will look at the other option here.

The pre- and post- variations both have the same overall effect—the two increment operators increment a variable and the two decrement operators decrement a variable—but they differ in terms of what they evaluate to. When you increment or decrement, there are really two values involved: the original value that the variable had before the increment or decrement operation, and the final value that the variable has after the increment or decrement operation. The post- versions evaluate to the original (older) value and the pre- versions evaluate to the final (later) value.

Consider, for example, the following code fragment:

```
int x = 10;
int y = 20;
int z = ++x * y--;
```

Continued on next page

*Continued from previous page*

What value is `z` assigned? The answer is 220. The third assignment increments `x` to 11 and decrements `y` to 19, but in computing the value of `z`, it uses the new value of `x` (`++x`) times the old value of `y` (`y--`), which is 11 times 20, or 220.


There is a simple mnemonic to remember this: When you see `x++`, read it as “give me `x`, then increment,” and when you see `++x`, read it as “increment, then give me `x`.” Another memory device that might help is to remember that C++ is a bad name for a programming language. The expression “C++” would be interpreted as “evaluate to the old value of C and then increment C.” In other words, even though you’re trying to come up with something new and different, you’re really stuck with the old awful language. The language you want is `++C`, which would be a new and improved language rather than the old one. Some people have suggested that perhaps Java is `++C`.

## Variables and Mixing Types

You already know that when you declare a variable, you must tell Java what type of value it will be storing. For example, you might declare a variable of type `int` for integer values or of type `double` for real values. The situation is fairly clear when you have just integers or just reals, but what happens when you start mixing the types? For example, the following code is clearly okay:

```
int x;
double y;
x = 2 + 3;
y = 3.4 * 2.9;
```

Here, we have an integer variable that we assign an integer value and a `double` variable that we assign a `double` value. But what if we try to do it the other way around?

```
int x;
double y;
 x = 3.4 * 2.9; // illegal
y = 2 + 3;      // okay
```

As the comments indicate, you can’t assign an integer variable a `double` value, but you can assign a `double` variable an integer value. Let’s consider the second case first. The expression `2 + 3` evaluates to the integer 5. This value isn’t a `double`, but every integer is a real value, so it is easy enough for Java to convert the integer into a `double`. The technical term is that Java *promotes* the integer into a `double`.

The first case is more problematic. The expression `3.4 * 2.9` evaluates to the `double` value 9.86. This value can’t be stored in an integer because it isn’t an integer. If you want to perform this kind of operation, you’ll have to tell Java to convert this



value into an integer. As described earlier, you can cast a double to an `int`, which will truncate anything after the decimal point:

```
x = (int) (3.4 * 2.9); // now legal
```

This statement first evaluates `3.4 * 2.9` to get `9.86` and then truncates that value to get the integer `9`.

### Common Programming Error

#### Forgetting to Cast

We often write programs that involve a mixture of `ints` and `doubles`, so it is easy to make mistakes when it comes to combinations of the two. For example, suppose that you want to compute the percentage of correctly answered questions on a student's test, given the total number of questions on the test and the number of questions the student got right. You might declare the following variables:

```
int totalQuestions;
int numRight;
double percent;
```

Suppose the first two are initialized as follows:

```
totalQuestions = 73;
numRight = 59;
```

How do you compute the percentage of questions that the student got right? You divide the number right by the total number of questions and multiply by 100 to turn it into a percentage:



```
percent = numRight / totalQuestions * 100; // incorrect
```

Unfortunately, if you print out the value of the variable `percent` after executing this line of code, you will find that it has the value `0.0`. But obviously the student got more than 0% correct.

The problem comes from integer division. The expression you are using begins with two `int` values:

```
numRight / totalQuestions
```

which means you are computing

```
59 / 73
```

*Continued on next page*

*Continued from previous page*

This evaluates to 0 with integer division. Some students fix this by changing the types of all the variables to `double`. That will solve the immediate problem, but it's not a good choice from a stylistic point of view. It is best to use the most appropriate type for data, and the number of questions on the test will definitely be an integer. You could try to fix this by changing the value 100 to 100.0:



```
percent = numRight / totalQuestions * 100.0; // incorrect
```

but this doesn't help because the division is done first. However, it does work if you put the 100.0 first:

```
percent = 100.0 * numRight / totalQuestions;
```

Now the multiplication is computed before the division, which means that everything is converted to `double`.

Sometimes you can fix a problem like this through a clever rearrangement of the formula, but you don't want to count on cleverness. This is a good place to use a cast. For example, returning to the original formula, you can cast each of the `int` variables to `double`:

```
percent = (double) numRight / (double) totalQuestions * 100.0;
```

You can also take advantage of the fact that once you have cast one of these two variables to `double`, the division will be done with doubles. So you could, for example, cast just the first value to `double`:

```
percent = (double) numRight / totalQuestions * 100.0;
```

## 2.3 The for Loop



VideoNote

Programming often involves specifying redundant tasks. The `for` loop helps to avoid such redundancy by repeatedly executing a sequence of statements over a particular range of values. Suppose you want to write out the squares of the first five integers. You could write a program like this:

```
1 public class WriteSquares {
2     public static void main(String[] args) {
3         System.out.println(1 + " squared = " + (1 * 1));
4         System.out.println(2 + " squared = " + (2 * 2));
5         System.out.println(3 + " squared = " + (3 * 3));
6         System.out.println(4 + " squared = " + (4 * 4));
7         System.out.println(5 + " squared = " + (5 * 5));
8     }
9 }
```

which would produce the following output:

```
1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16
5 squared = 25
```

But this approach is tedious. The program has five statements that are very similar. They are all of the form:

```
System.out.println(number + " squared = " + (number * number));
```

where `number` is either 1, 2, 3, 4, or 5. The `for` loop avoids such redundancy. Here is an equivalent program using a `for` loop:

```
1 public class WriteSquares2 {
2     public static void main(String[] args) {
3         for (int i = 1; i <= 5; i++) {
4             System.out.println(i + " squared = " + (i * i));
5         }
6     }
7 }
```

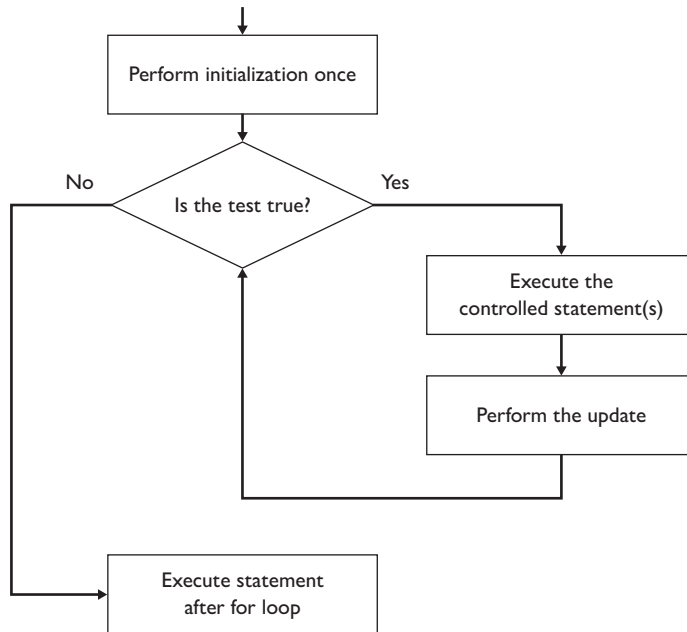
This program initializes a variable called `i` to the value 1. Then it repeatedly executes the `println` statement as long as the variable `i` is less than or equal to 5. After each `println`, it evaluates the expression `i++` to increment `i`.

The general syntax of the `for` loop is as follows:

```
for (<initialization>; <continuation test>; <update>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

You always include the keyword `for` and the parentheses. Inside the parentheses are three different parts, separated by semicolons: the initialization, the continuation test, and the update. Then there is a set of curly braces that encloses a set of statements. The `for` loop controls the statements inside the curly braces. We refer to the controlled statements as the *body* of the loop. The idea is that we execute the body multiple times, as determined by the combination of the other three parts.

The diagram in Figure 2.1 indicates the steps that Java follows to execute a `for` loop. It performs whatever initialization you have requested once before the loop begins executing. Then it repeatedly performs the continuation test you have provided.



**Figure 2.1** Flow of for loop

If the continuation test evaluates to `true`, it executes the controlled statements once and executes the update part. Then it performs the test again. If it again evaluates to `true`, it executes the statements again and executes the update again. Notice that the update is performed after the controlled statements are executed. When the test evaluates to `false`, Java is done executing the loop and moves on to whatever statement comes after the loop.

The `for` loop is the first example of a *control structure* that we will study.

### Control Structure

A syntactic structure that controls other statements.

You should be careful to use indentation to indicate controlled statements. In the case of the `for` loop, all of the statements in the body of the loop are indented as a way to indicate that they are “inside” the loop.

### Tracing for Loops

Let’s examine the `for` loop of the `writeSquares2` program in detail:

```

for (int i = 1; i <= 5; i++) {
    System.out.println(i + " squared = " + (i * i));
}

```

Table 2.6 Trace of for (int i = 1; i <= 5; i++)

Step	Code	Description
initialization	int i = 1;	variable i is created and initialized to 1
test	i <= 5	true because 1 <= 5, so we enter the loop
body	{ . . . }	execute the println with i equal to 1
update	i++	increment i, which becomes 2
test	i <= 5	true because 2 <= 5, so we enter the loop
body	{ . . . }	execute the println with i equal to 2
update	i++	increment i, which becomes 3
test	i <= 5	true because 3 <= 5, so we enter the loop
body	{ . . . }	execute the println with i equal to 3
update	i++	increment i, which becomes 4
test	i <= 5	true because 4 <= 5, so we enter the loop
body	{ . . . }	execute the println with i equal to 4
update	i++	increment i, which becomes 5
test	i <= 5	true because 5 <= 5, so we enter the loop
body	{ . . . }	execute the println with i equal to 5
update	i++	increment i, which becomes 6
test	i <= 5	false because 6 > 5, so we are finished

In this loop, the initialization (`int i = 1`) declares an integer variable `i` that is initialized to 1. The continuation test (`i <= 5`) indicates that we should keep executing as long as `i` is less than or equal to 5. That means that once `i` is greater than 5, we will stop executing the body of the loop. The update (`i++`) will increment the value of `i` by one each time, bringing `i` closer to being larger than 5. After five executions of the body and the accompanying five updates, `i` will be larger than 5 and the loop will finish executing. Table 2.6 traces this process in detail.

Java allows great flexibility in deciding what to include in the initialization part and the update, so we can use the `for` loop to solve all sorts of programming tasks. For now, though, we will restrict ourselves to a particular kind of loop that declares and initializes a single variable that is used to control the loop. This variable is often referred to as the *control variable* of the loop. In the test we compare the control variable against some final desired value, and in the update we change the value of the control variable, most often incrementing it by 1. Such loops are very common in programming. By convention, we often use names like `i`, `j`, and `k` for the control variables.

Each execution of the controlled statement of a loop is called an *iteration* of the loop (as in, “The loop finished executing after four iterations”). Iteration also refers to looping in general (as in, “I solved the problem using iteration”).

Consider another `for` loop:

```
for (int i = 100; i <= 100; i++) {  
    System.out.println(i + " squared = " + (i * i));  
}
```

This loop executes a total of 201 times, producing the squares of all the integers between  $-100$  and  $+100$  inclusive. The values used in the initialization and the test, then, can be any integers. They can, in fact, be arbitrary integer expressions:

```
for (int i = (2 + 2); i <= (17 * 3); i++) {
    System.out.println(i + " squared = " + (i * i));
}
```

This loop will generate the squares of all the integers between 4 and 51 inclusive. The parentheses around the expressions are not necessary but improve readability. Consider the following loop:

```
for (int i = 1; i <= 30; i++) {
    System.out.println("+-----+");
}
```

This loop generates 30 lines of output, all exactly the same. It is slightly different from the previous one because the statement controlled by the `for` loop makes no reference to the control variable. Thus,

```
for (int i = -30; i <= -1; i++) {
    System.out.println("+-----+");
}
```

generates exactly the same output. The behavior of such a loop is determined solely by the number of iterations it performs. The number of iterations is given by

$$\text{<ending value>} - \text{<starting value>} + 1$$

It is much simpler to see that the first of these loops iterates 30 times, so it is better to use that loop.

Now let's look at some borderline cases. Consider this loop:

```
for (int i = 1; i <= 1; i++) {
    System.out.println("+-----+");
}
```

According to our rule it should iterate once, and it does. It initializes the variable `i` to 1 and tests to see if this is less than or equal to 1, which it is. So it executes the `println`, increments `i`, and tests again. The second time it tests, it finds that `i` is no longer less than or equal to 1, so it stops executing. Now consider this loop:

```
for (int i = 1; i <= 0; i++) {
    System.out.println("+-----+"); // never executes
}
```

This loop performs no iterations at all. It will not cause an execution error; it just won't execute the body. It initializes the variable to 1 and tests to see if this is less than or equal to 0. It isn't, so rather than executing the statements in the body, it stops there.

When you construct a `for` loop, you can include more than one statement inside the curly braces. Consider, for example, the following code:

```
for (int i = 1; i <= 20; i++) {  
    System.out.println("Hi!");  
    System.out.println("Ho!");  
}
```

This will produce 20 pairs of lines, the first of which has the word “Hi!” on it and the second of which has the word “Ho!”

When a `for` loop controls a single statement, you don't have to include the curly braces. The curly braces are required only for situations like the previous one, where you have more than one statement that you want the loop to control. However, the Java coding convention includes the curly braces even for a single statement, and we follow this convention in this book. There are two advantages to this convention:

- Including the curly braces prevents future errors. Even if you need only one statement in the body of your loop now, your code is likely to change over time. Having the curly braces there ensures that, if you add an extra statement to the body later, you won't accidentally forget to include them. In general, including curly braces in advance is cheaper than locating obscure bugs later.
- Always including the curly braces reduces the level of detail you have to consider as you learn new control structures. It takes time to master the details of any new control structure, and it will be easier to master those details if you don't have to also be thinking about when to include and when not to include the braces.

### Common Programming Error

#### Forgetting Curly Braces

You should use indentation to indicate the body of a `for` loop, but indentation alone is not enough. Java ignores indentation when it is deciding how different statements are grouped. Suppose, for example, that you were to write the following code:

```
for (int i = 1; i <= 20; i++)  
    System.out.println("Hi!");  
    System.out.println("Ho!");
```

The indentation indicates to the reader that both of the `println` statements are in the body of the `for` loop, but there aren't any curly braces to indicate that to Java. As a result, this code is interpreted as follows:

*Continued on next page*

*Continued from previous page*

```
for (int i = 1; i <= 20; i++) {  
    System.out.println("Hi!");  
}  
System.out.println("Ho!");
```

Only the first `println` is considered to be in the body of the `for` loop. The second `println` is considered to be outside the loop. So, this code would produce 20 lines of output that all say “Hi!” followed by one line of output that says “Ho!” To include both `println`s in the body, you need curly braces around them:

```
for (int i = 1; i <= 20; i++) {  
    System.out.println("Hi!");  
    System.out.println("Ho!");  
}
```

## for Loop Patterns

In general, if you want a loop to iterate exactly  $n$  times, you will use one of two standard loops. The first standard form looks like the ones you have already seen:

```
for (int <variable> = 1; <variable> <= n; i++) {  
    <statement>;  
    <statement>;  
    ...  
    <statement>;  
}
```

It’s pretty clear that this loop executes  $n$  times, because it starts at 1 and continues as long as it is less than or equal to  $n$ . For example, this loop prints the numbers 1 through 10:

```
for (int i = 1; i <= 10; i++) {  
    System.out.print(i + " ");  
}
```

Because it uses a `print` instead of a `println` statement, it produces a single line of output:

```
1 2 3 4 5 6 7 8 9 10
```

Often, however, it is more convenient to start our counting at 0 instead of 1. That requires a change in the loop test to allow you to stop when  $n$  is one less:



```

for (int <variable> = 0; <variable> < n; i++) {
    <statement>;
    <statement>;
    ...
    <statement>;
}

```

Notice that in this form when you initialize the variable to 0, you test whether it is strictly less than  $n$ . Either form will execute exactly  $n$  times, although there are some situations where the zero-based loop works better. For example, this loop executes 10 times just like the previous loop:

```

for (int i = 0; i < 10; i++) {
    System.out.print(i + " ");
}

```

Because it starts at 0 instead of starting at 1, it produces a different sequence of 10 values:

```
0 1 2 3 4 5 6 7 8 9
```

Most often you will use the loop that starts at 0 or 1 to perform some operation a fixed number of times. But there is a slight variation that is also sometimes useful. Instead of running the loop in a forward direction, we can run it backward. Instead of starting at 1 and executing until you reach  $n$ , you instead start at  $n$  and keep executing until you reach 1. You can accomplish this by using a decrement rather than an increment, so we sometimes refer to this as a decrementing loop.

Here is the general form of a decrementing loop:

```

for (int <variable> = n; <variable> >= 1; <variable>—) {
    <statement>;
    <statement>;
    ...
    <statement>;
}

```

For example, here is a decrementing loop that executes 10 times:

```

for (int i = 10; i >= 1; i—) {
    System.out.print(i + " ");
}

```

Because it runs backward, it prints the values in reverse order:

```
10 9 8 7 6 5 4 3 2 1
```