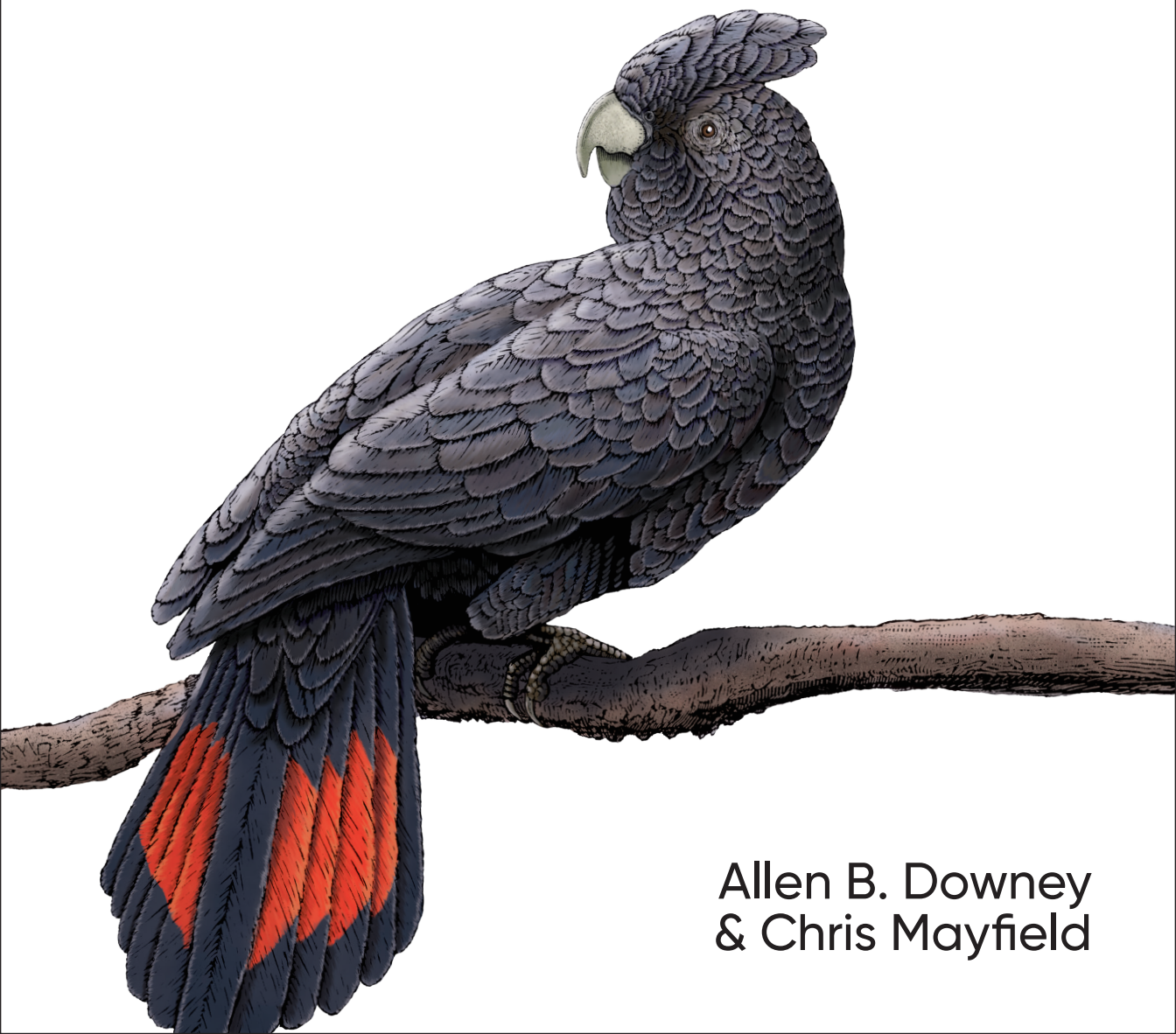


O'REILLY®

Second
Edition

Think Java

How to Think Like a Computer Scientist



Allen B. Downey
& Chris Mayfield

Think Java

Think Java is a hands-on introduction to computer science and programming used by many universities and high schools around the world. Its conciseness, emphasis on vocabulary, and informal tone make it particularly appealing for readers with little or no experience. The book starts with the most basic programming concepts and gradually works its way to advanced object-oriented techniques.

In this fully updated and expanded edition, authors Allen Downey and Chris Mayfield introduce programming as a means for solving interesting problems. Each chapter presents material for one week of a college course and includes exercises to help you practice what you've learned. Along the way, you'll see nearly every topic required for the AP Computer Science A exam and Java SE Programmer I certification.

- Discover one concept at a time: tackle complex topics in a series of small steps with multiple examples
- Understand how to formulate problems, think creatively about solutions, and develop, test, and debug programs
- Learn about input and output, decisions and loops, classes and methods, strings and arrays, recursion and polymorphism
- Determine which program development methods work best for you, and practice the important skill of debugging

Allen Downey is a professor of computer science at Olin College of Engineering. He has a PhD in computer science from UC Berkeley and master's and bachelor's degrees from MIT.

Chris Mayfield is an associate professor of computer science at James Madison University. He has a PhD in computer science from Purdue and two bachelor's degrees from the University of Utah.

*"With a strong emphasis on problem solving, *Think Java* moves beyond just teaching coding and really delves into the underlying concepts of computer science. It is a great book to move students from beginners to thinking like computer scientists."*

—Rebecca Dovi
CodeVA

"I like the book's concise approach that emphasizes critical thinking and problem-solving skills, with enough specifics on the Java language to enable students to practice the art of programming."

—David Wisneski
CSUMB

JAVA/BEGINNING PROGRAMMING

US \$49.99

CAN \$65.99

ISBN: 978-1-492-07250-8



Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

Think Java

How to Think Like a Computer Scientist

Allen B. Downey and Chris Mayfield

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY[®]

Think Java

by Allen B. Downey and Chris Mayfield

Copyright © 2020 Allen B. Downey and Chris Mayfield. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Suzanne McQuade

Development Editor: Corbin Collins

Production Editor: Christopher Faucher

Copyeditor: Sharon Wilkey

Proofreader: Christina Edwards

Indexers: Allen Downey and Chris Mayfield

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

May 2016: First Edition
December 2019: Second Edition

Revision History for the Second Edition

2019-11-27: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492072508> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Think Java*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

Think Java is available under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. The authors maintain an online version at <https://greenteapress.com/wp/think-java-2e>.

978-1-492-07250-8

[LSI]

Table of Contents

Preface	ix
1. Computer Programming	1
What Is a Computer?	1
What Is Programming?	2
The Hello World Program	3
Compiling Java Programs	4
Displaying Two Messages	6
Formatting Source Code	6
Using Escape Sequences	7
What Is Computer Science?	8
Debugging Programs	9
Vocabulary	10
Exercises	12
2. Variables and Operators	15
Declaring Variables	15
Assigning Variables	16
Memory Diagrams	17
Printing Variables	18
Arithmetic Operators	19
Floating-Point Numbers	20
Rounding Errors	21
Operators for Strings	22
Compiler Error Messages	23
Other Types of Errors	24
Vocabulary	25
Exercises	27

3. Input and Output	29
The System Class	29
The Scanner Class	30
Language Elements	31
Literals and Constants	32
Formatting Output	34
Reading Error Messages	35
Type Cast Operators	36
Remainder Operator	37
Putting It All Together	38
The Scanner Bug	39
Vocabulary	40
Exercises	41
4. Methods and Testing	45
Defining New Methods	45
Flow of Execution	46
Parameters and Arguments	47
Multiple Parameters	49
Stack Diagrams	50
Math Methods	51
Composition	52
Return Values	53
Incremental Development	54
Vocabulary	56
Exercises	57
5. Conditionals and Logic	63
Relational Operators	63
The if-else Statement	64
Chaining and Nesting	65
The switch Statement	66
Logical Operators	67
De Morgan's Laws	69
Boolean Variables	69
Boolean Methods	70
Validating Input	71
Example Program	72
Vocabulary	73
Exercises	74

6. Loops and Strings.....	79
The while Statement	79
Increment and Decrement	81
The for Statement	81
Nested Loops	83
Characters	84
Which Loop to Use	85
String Iteration	86
The indexOf Method	87
Substrings	87
String Comparison	88
String Formatting	89
Vocabulary	90
Exercises	91
7. Arrays and References.....	95
Creating Arrays	96
Accessing Elements	97
Displaying Arrays	98
Copying Arrays	99
Traversing Arrays	100
Generating Random Numbers	102
Building a Histogram	103
The Enhanced for Loop	104
Counting Characters	105
Vocabulary	107
Exercises	108
8. Recursive Methods.....	111
Recursive Void Methods	111
Recursive Stack Diagrams	113
Value-Returning Methods	114
The Leap of Faith	116
Counting Up Recursively	117
Binary Number System	118
Recursive Binary Method	119
CodingBat Problems	120
Vocabulary	122
Exercises	123
9. Immutable Objects.....	129
Primitives Versus Objects	129

The null Keyword	130
Strings Are Immutable	131
Wrapper Classes	132
Command-Line Arguments	134
Argument Validation	135
BigInteger Arithmetic	136
Incremental Design	137
More Generalization	139
Vocabulary	140
Exercises	141
10. Mutable Objects.....	147
Point Objects	147
Objects as Parameters	148
Objects as Return Values	149
Rectangles Are Mutable	150
Aliasing Revisited	151
Java Library Source	152
Class Diagrams	153
Scope Revisited	154
Garbage Collection	154
Mutable Versus Immutable	155
StringBuilder Objects	156
Vocabulary	157
Exercises	158
11. Designing Classes.....	161
The Time Class	161
Constructors	162
Value Constructors	164
Getters and Setters	165
Displaying Objects	167
The toString Method	167
The equals Method	168
Adding Times	170
Vocabulary	172
Exercises	173
12. Arrays of Objects.....	175
Card Objects	175
Card toString	177
Class Variables	178

The compareTo Method	179
Cards Are Immutable	180
Arrays of Cards	181
Sequential Search	183
Binary Search	183
Tracing the Code	184
Vocabulary	185
Exercises	186
13. Objects of Arrays.....	189
Decks of Cards	189
Shuffling Decks	190
Selection Sort	192
Merge Sort	192
Subdecks	193
Merging Decks	194
Adding Recursion	195
Static Context	196
Piles of Cards	197
Playing War	199
Vocabulary	200
Exercises	201
14. Extending Classes.....	205
CardCollection	206
Inheritance	208
Dealing Cards	209
The Player Class	211
The Eights Class	212
Class Relationships	215
Vocabulary	216
Exercises	216
15. Arrays of Arrays.....	219
Conway's Game of Life	219
The Cell Class	221
Two-Dimensional Arrays	222
The GridCanvas Class	223
Other Grid Methods	224
Starting the Game	225
The Simulation Loop	226
Exception Handling	227

Counting Neighbors	227
Updating the Grid	229
Vocabulary	231
Exercises	231
16. Reusing Classes.....	235
Langton's Ant	235
Refactoring	237
Abstract Classes	238
UML Diagram	240
Vocabulary	241
Exercises	241
17. Advanced Topics.....	243
Polygon Objects	243
Adding Color	244
Regular Polygons	245
More Constructors	247
An Initial Drawing	248
Blinking Polygons	250
Interfaces	251
Event Listeners	253
Timers	256
Vocabulary	257
Exercises	258
A. Tools.....	259
B. Javadoc.....	271
C. Graphics.....	279
D. Debugging.....	287
Index.....	299

Preface

Think Java is an introduction to computer science and programming intended for readers with little or no experience. We start with the most basic concepts and are careful to define all terms when they are first used. The book presents each new idea in a logical progression. Larger topics, like control flow statements and object-oriented programming, are divided into smaller examples and introduced over the course of several chapters.

This book is intentionally concise. Each chapter is 10–12 pages and covers the material for one week of a college course. It is not meant to be a comprehensive presentation of Java, but rather, an initial exposure to programming constructs and techniques. We begin with small problems and basic algorithms and work up to object-oriented design. In the vocabulary of computer science pedagogy, this book uses the *objects late* approach.

The Philosophy Behind the Book

Here are the guiding principles that make the book the way it is:

One concept at a time

We break down topics that give beginners trouble into a series of small steps, so that they can exercise each new concept in isolation before continuing.

Balance of Java and concepts

The book is not primarily about Java; it uses code examples to demonstrate computer science. Most chapters start with language features and end with concepts.

Conciseness

An important goal of the book is to be small enough so that students can read and understand the entire text in a one-semester college or AP course.

Emphasis on vocabulary

We try to introduce the minimum number of terms and define them carefully when they are first used. We also organize them in glossaries at the end of each chapter.

Program development

There are many strategies for writing programs, including bottom-up, top-down, and others. We demonstrate multiple program development techniques, allowing readers to choose methods that work best for them.

Multiple learning curves

To write a program, you have to understand the algorithm, know the programming language, and be able to debug errors. We discuss these and other aspects throughout the book, and summarize our advice in [Appendix D](#).

Object-Oriented Programming

Some Java books introduce classes and objects immediately; others begin with procedural programming and transition to object-oriented more gradually.

Many of Java's object-oriented features are motivated by problems with previous languages, and their implementations are influenced by this history. Some of these features are hard to explain when people aren't familiar with the problems they solve.

We get to object-oriented programming as quickly as possible (beginning with [Chapter 9](#)). But we introduce concepts one at a time, as clearly as possible, in a way that allows readers to practice each idea in isolation before moving on. So it takes some time to get there.

You can't write Java programs (even Hello World) without encountering object-oriented features. In some cases we explain a feature briefly when it first appears, and then explain it more deeply later on.

If you read the entire book, you will see nearly every topic required for Java SE Programmer I certification. Supplemental lessons are available in the official Java tutorials on <http://thinkjava.org/tutorial>.

This book is also well suited to prepare high school students for the AP Computer Science A exam, which includes object-oriented design and implementation. (AP is a registered trademark of The College Board.) A mapping of *Think Java* section numbers to the AP course is available on <https://thinkjava.org>.

Changes to the Second Edition

This new edition was written over several years, with feedback from dozens of instructors and hundreds of students. A complete history of all changes is available on GitHub. Here are some of the highlights:

Chapters 2–4

We reordered the material in Chapter 1 to present a more interesting balance of theory and practice. Chapters 2 and 3 are much cleaner now too. Methods are now presented in a single chapter, along with additional in-depth examples.

Chapters 5–8

We rearranged these chapters a lot, added many examples and new figures, and removed unnecessary details. Strings are covered earlier (before arrays) so that readers can apply them to loop problems. The material on recursion is now a chapter, and we added new sections to explain binary numbers and CodingBat.

Chapters 9–12

Our main goal for these chapters was to provide better explanations and more diagrams. Chapters 9 and 10 focus more on immutable versus mutable objects, and we added new sections on `BigInteger` and `StringBuilder`. The other content is largely the same, but it should be easier to understand now.

Chapters 13–17

We balanced the amount of content in Chapters 13–14 by moving `ArrayLists` earlier, and we implement the War card game as another example. Chapters 15–17 are brand-new in this edition; they cover more advanced topics including 2D arrays, graphics, exceptions, abstract classes, interfaces, and events.

Appendixes

We added Appendix B to explain documentation comments and Javadoc in more detail. The other three appendixes that were present in the first edition have been revised for clarity and layout.

About the Appendixes

The chapters of this book are meant to be read in order, because each one builds on the previous one. We also include several appendixes with material that can be read at any time:

Appendix A, “Tools”

This appendix explains how to download and install Java so you can compile programs on your computer. It also provides a brief introduction to DrJava—an *integrated development environment* (IDE) that is designed primarily for students

—and other development tools, including Checkstyle for code quality and JUnit for testing.

Appendix B, “Javadoc”

It’s important to document your classes and methods so that other programmers (including yourself in the future) will know how to use them. This appendix explains how to read documentation, how to write documentation, and how to use the Javadoc tool.

Appendix C, “Graphics”

Java provides libraries for working with graphics and animation, and these topics can be engaging for students. The libraries require object-oriented features that students will not completely understand until after Chapter 10, but they can be used much earlier.

Appendix D, “Debugging”

We provide debugging suggestions throughout the book, but this appendix provides many more suggestions on how to debug your programs. We recommend that you review this appendix frequently as you work through the book.

Using the Code Examples

Most of the code examples in this book are available from <https://github.com/ChrisMayfield/ThinkJavaCode2>. Git is a “version control system” that allows you to keep track of the files that make up a project. A collection of files under Git’s control is called a *repository*.

GitHub is a hosting service that provides storage for Git repositories and a convenient web interface. It provides several ways to work with the code:

- You can create a copy of the repository on GitHub by clicking the Fork button. If you don’t already have a GitHub account, you’ll need to create one. After forking, you’ll have your own repository on GitHub that you can use to keep track of code you write. Then you can *clone* the repository, which downloads a copy of the files to your computer.
- Alternatively, you could clone the original repository without forking. If you choose this option, you don’t need a GitHub account, but you won’t be able to save your changes on GitHub.
- If you don’t want to use Git at all, you can download the code in a ZIP archive using the Download ZIP button on the GitHub page, or this link: <https://thinkjava.org/code2zip>.

After you clone the repository or unzip the ZIP file, you should have a directory named *ThinkJavaCode2* with a subdirectory for each chapter in the book.

The examples in this book were developed and tested using OpenJDK 11. If you are using a more recent version, everything should still work. If you are using an older version, some of the examples might not.

If example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Think Java* by Allen B. Downey and Chris Mayfield. (O'Reilly). Copyright 2020 Allen B. Downey and Chris Mayfield, 978-1-492-07250-8.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Bold

Indicates vocabulary words defined at the end of each chapter.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

O'Reilly Online Learning

O'REILLY® For more than 40 years, **O'Reilly Media** has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/think-java-2e>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

Many people have sent corrections and suggestions over the years, and we appreciate their valuable feedback! This list begins with version 4.0 of the open source edition, so it omits those who contributed to earlier versions:

- Ellen Hildreth used this book to teach data structures at Wellesley College and submitted a whole stack of corrections and suggestions.
- Tania Passfield pointed out that some glossaries had leftover terms that no longer appeared in the text.
- Elizabeth Wiethoff noticed that the series expansion of $\exp(-x^2)$ was wrong. She has also worked on a Ruby version of the book.
- Matt Crawford sent in a whole patch file full of corrections.
- Chi-Yu Li pointed out a typo and an error in one of the code examples.
- Doan Thanh Nam corrected an example.
- Muhammad Saied translated the book into Arabic and found several errors in the process.
- Marius Margowski found an inconsistency in a code example.
- Leslie Klein discovered another error in the series expansion of $\exp(-x^2)$, identified typos in the card array figures, and helped clarify several exercises.
- Micah Lindstrom reported half a dozen typos and sent corrections.
- James Riely ported the textbook source from LaTeX to <http://fp.cs.depaul.edu/jriely/thinkajava>.
- Peter Knaggs ported the book to <https://www.rigwit.co.uk/think/sharp>.
- Heidi Gentry-Kolen recorded several <https://www.youtube.com/user/digipipeline> that follow the book.
- Waldo Ribeiro submitted a pull request that corrected a dozen typos.
- Michael Stewart made suggestions for improving the first half of the book.
- Steven Richardson adapted the book for an online course and contributed many ideas for improving the text.
- Fazl Rahman provided detailed feedback, chapter by chapter, and offered many suggestions for improving the text.

We are especially grateful to the technical reviewers of the O'Reilly Media first edition: Blythe Samuels, David Wisneski, and Stephen Rose. They found errors, made many great suggestions, and helped make the book much better.

Likewise, we thank Marc Loy for his thorough review of the O'Reilly Media second edition. He contributed many corrections, insights, and clarifications.

Many students have given exceptional feedback, including Ian Staton, Tanner Wernecke, Jacob Green, Rasha Abuhantash, Nick Duncan, Kylie Davidson, Shirley Jiang, Elena Trafton, Jennifer Gregorio, and Azeem Mufti.

Other contributors who found one or more typos: Stijn Debrouwere, Guy Driesen, Andai Velican, Chris Kuzmaul, Daniel Kurikesu, Josh Donath, Rens Findhammer, Elisa Abedrapo, Yousef BaAfif, Bruce Hill, Matt Underwood, Isaac Sultan, Dan Rice, Robert Beard, Daniel Pierce, Michael Gifftthaler, Chris Fox, Min Zeng, Markus Geuss, Mauricio Gonzalez, Enrico Sartirana, Kasem Satitwiwat, and Jason Miller.

If you have additional comments or ideas about the text, please send them to feedback@greenteapress.com.

—Allen Downey and Chris Mayfield

Computer Programming

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas—specifically, computations. Like engineers, they design things, assembling components into systems and evaluating trade-offs among alternatives. And like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

An important skill for a computer scientist is **problem solving**. It involves the ability to formulate problems, think creatively about solutions, and express solutions clearly and accurately. As it turns out, the process of learning to program computers is an excellent opportunity to develop problem-solving skills. On one level, you will be learning to write Java programs, a useful skill by itself. But on another level, you will use programming as a means to an end. As we go along, that end will become clearer.

What Is a Computer?

When people hear the word *computer*, they often think of a desktop or laptop. Not surprisingly, searching for “computer” on [Google Images](#) displays rows and rows of these types of machines. However, in a more general sense, a computer can be any type of device that stores and processes data.

Dictionary.com defines a computer as “a programmable electronic device designed to accept data, perform prescribed mathematical and logical operations at high speed, and display the results of these operations. Mainframes, desktop and laptop computers, tablets, and smartphones are some of the different types of computers.”

Each type of computer has its own unique design, but internally they all share the same type of **hardware**. The two most important hardware components are

processors (or CPUs) that perform simple calculations and **memory** (or RAM) that temporarily stores information. **Figure 1-1** shows what these components look like.



Figure 1-1. Example processor and memory hardware

Users generally see and interact with touchscreens, keyboards, and monitors, but it's the processors and memory that perform the actual computation. Nowadays it's fairly standard, even for a smartphone, to have at least eight processors and four gigabytes (four billion cells) of memory.

What Is Programming?

A **program** is a sequence of instructions that specifies how to perform a computation on computer hardware. The computation might be something mathematical, like solving a system of equations or finding the roots of a polynomial. It could also be a symbolic computation, like searching and replacing text in a document or (strangely enough) compiling a program.

The details look different in different languages, but a few basic instructions appear in just about every language:

Input

Get data from the keyboard, a file, a sensor, or some other device.

Output

Display data on the screen, or send data to a file or other device.

Math

Perform basic mathematical operations like addition and division.

Secision

Check for certain conditions and execute the appropriate code.

Repetition

Perform an action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of small instructions that look much like these. So you can think of **programming** as the process of breaking a large, complex task into smaller and smaller subtasks. The process continues until the subtasks are simple enough to be performed with the electronic circuits provided by the hardware.

The Hello World Program

Traditionally, the first program you write when learning a new programming language is called the *Hello World program*. All it does is output the words Hello, World! to the screen. In Java, it looks like this:

```
public class Hello {  
  
    public static void main(String[] args) {  
        // generate some simple output  
        System.out.println("Hello, World!");  
    }  
}
```

When this program runs, it displays the following:

```
Hello, World!
```

Notice that the output does not include the quotation marks.

Java programs are made up of *class* and *method* definitions, and methods are made up of *statements*. A **statement** is a line of code that performs a basic action. In the Hello World program, this line is a **print statement** that displays a message to the user:

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

`System.out.println` displays results on the screen; the name `println` stands for *print line*. Confusingly, *print* can mean both *display on the screen* and *send to the printer*. In this book, we'll try to say *display* when we mean output to the screen. Like most statements, the print statement ends with a semicolon (;).

Java is *case-sensitive*, which means that uppercase and lowercase are not the same. In the Hello World program, `System` has to begin with an uppercase letter; `system` and `SYSTEM` won't work.

A **method** is a named sequence of statements. This program defines one method named `main`:

```
public static void main(String[] args)
```

The name and format of `main` is special: when the program runs, it starts at the first statement in `main` and ends when it finishes the last statement. Later, you will see programs that define more than one method.

This program defines a class named `Hello`. For now, a **class** is a collection of methods; we'll have more to say about this later. You can give a class any name you like, but it is conventional to start with a capital letter. The name of the class has to match the name of the file it is in, so this class has to be in a file named *Hello.java*.

Java uses curly braces (`{` and `}`) to group things together. In *Hello.java*, the outermost braces contain the class definition, and the inner braces contain the method definition.

The line that begins with two slashes (`//`) is a **comment**, which is a bit of English text that explains the code. When Java sees `//`, it ignores everything from there until the end of the line. Comments have no effect on the execution of the program, but they make it easier for other programmers (and your future self) to understand what you meant to do.

Compiling Java Programs

The programming language you will learn in this book is Java, which is a **high-level language**. Other high-level languages you may have heard of include Python, C and C++, PHP, Ruby, and JavaScript.

Before they can run, programs in high-level languages have to be translated into a **low-level language**, also called *machine language*. This translation takes some time, which is a small disadvantage of high-level languages. But high-level languages have two major advantages:

- It is *much* easier to program in a high-level language. Programs take less time to write, are shorter and easier to read, and are more likely to be correct.
- High-level languages are **portable**, meaning they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer.

Two kinds of programs translate high-level languages into low-level languages: interpreters and compilers. An **interpreter** reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations. [Figure 1-2](#) shows the structure of an interpreter.

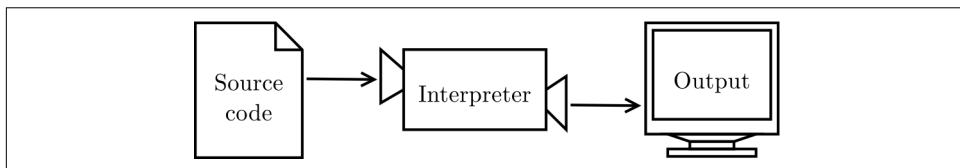


Figure 1-2. How interpreted languages are executed

In contrast, a **compiler** reads the entire program and translates it completely before the program starts running. The high-level program is called the **source code**. The translated program is called the **object code**, or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation. As a result, compiled programs often run faster than interpreted programs.

Note that object code, as a low-level language, is not portable. You cannot run an executable compiled for a Windows laptop on an Android phone, for example. To run a program on different types of machines, it must be compiled multiple times. It can be difficult to write source code that compiles and runs correctly on different types of machines.

To address this issue, Java is *both* compiled and interpreted. Instead of translating source code directly into an executable, the Java compiler generates code for a **virtual machine**. This “imaginary” machine has the functionality common to desktops, laptops, tablets, phones, etc. Its language, called Java **byte code**, looks like object code and is easy and fast to interpret.

As a result, it’s possible to compile a Java program on one machine, transfer the byte code to another machine, and run the byte code on that other machine. [Figure 1-3](#) shows the steps of the development process. The Java compiler is a program named `javac`. It translates `.java` files into `.class` files that store the resulting byte code. The Java interpreter is another program, named `java`, which is short for *Java Virtual Machine (JVM)*.

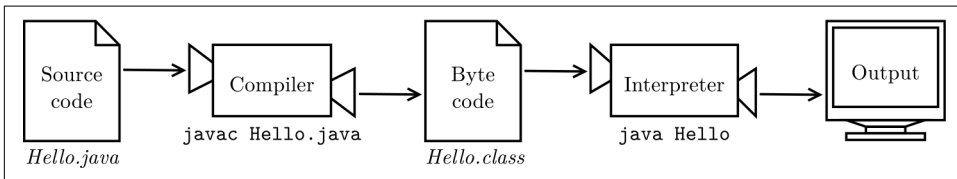


Figure 1-3. The process of compiling and running a Java program

The programmer writes source code in the file `Hello.java` and uses `javac` to compile it. If there are no errors, the compiler saves the byte code in the file `Hello.class`. To run the program, the programmer uses `java` to interpret the byte code. The result of the program is then displayed on the screen.

Although it might seem complicated, these steps are automated for you in most development environments. Usually, you have to only press a button or type a single command to compile and interpret your program. On the other hand, it is important to know what steps are happening in the background, so if something goes wrong, you can figure out what it is.

Displaying Two Messages

You can put as many statements as you like in the `main` method. For example, to display more than one line of output:

```
public class Hello2 {  
  
    public static void main(String[] args) {  
        // generate some simple output  
        System.out.println("Hello, World!"); // first line  
        System.out.println("How are you?"); // another line  
    }  
}
```

As this example also shows, you can put comments at the end of a line as well as on lines all by themselves.

Phrases that appear in quotation marks are called **strings**, because they contain a sequence of characters strung together in memory. Characters can be letters, numbers, punctuation marks, symbols, spaces, tabs, etc.

`System.out.println` appends a special character, called a **newline**, that moves to the beginning of the next line. If you don't want a newline at the end, you can use `print` instead of `println`:

```
public class Goodbye {  
  
    public static void main(String[] args) {  
        System.out.print("Goodbye, ");  
        System.out.println("cruel world");  
    }  
}
```

In this example, the first statement does not add a newline, so the output appears on a single line:

```
Goodbye, cruel world
```

Notice that there is a space at the end of the first string, which appears in the output just before the word `cruel`.

Formatting Source Code

In Java source code, some spaces are required. For example, you need at least one space between words, so this program is not legal:


```
public class Goodbye {
    public static void main(String[] args) {
        System.out.print("Goodbye, ");
        System.out.println("cruel world");
    }
}
```

But most other spaces are optional. For example, this program *is* legal:

```
public class Goodbye {
public static void main(String[] args) {
System.out.print("Goodbye, ");
System.out.println("cruel world");
}
}
```

The newlines are optional, too. So we could just write this:

```
public class Goodbye { public static void main(String[] args)
{ System.out.print("Goodbye, "); System.out.println
("cruel world");}}
```

It still works, but the program is getting harder and harder to read. Newlines and spaces are important for visually organizing your program, making it easier to understand the program and find errors when they occur.

Many editors will automatically format source code with consistent indenting and line breaks. For example, in DrJava (see [“Installing DrJava” on page 259](#)) you can indent your code by selecting all text (Ctrl+A) and pressing the Tab key.

Organizations that do a lot of software development usually have strict guidelines on how to format source code. For example, Google publishes its [Java coding standards](#) for use in open source projects.

You probably won’t understand these guidelines now, because they refer to language features you haven’t yet seen. But you might want to refer to them periodically as you read this book.

Using Escape Sequences

It’s possible to display multiple lines of output with only one line of code. You just have to tell Java where to put the line breaks:

```
public class Hello3 {
    public static void main(String[] args) {
        System.out.print("Hello!\nHow are you doing?\n");
    }
}
```

The output is two lines, each ending with a newline character:

```
Hello!  
How are you doing?
```

Each `\n` is an **escape sequence**, or two characters of source code that represent a single character. (The backslash allows you to *escape* the string to write special characters.) Notice there is no space between `\n` and `How`. If you add a space there, there will be a space at the beginning of the second line. Java has a total of eight escape sequences, and the four most commonly used ones are listed in [Table 1-1](#).

Table 1-1. Common escape sequences

<code>\n</code>	Newline
<code>\t</code>	Tab
<code>\"</code>	Double quote
<code>\\</code>	Backslash

For example, to write quotation marks inside of strings, you need to escape them with a backslash:

```
System.out.println("She said \"Hello!\" to me.");
```

The result is as follows:

```
She said "Hello!" to me.
```

What Is Computer Science?

This book intentionally omits some details about the Java language (such as the other escape sequences), because our main goal is teaching you how to think like a computer scientist. Being able to understand computation is much more valuable than just learning how to write code.

If you're interested in learning more about Java itself, Oracle maintains an official set of tutorials on its [website](#). The “Language Basics” tutorial (found under “Learning the Java Language”) is a good place to start.

One of the most interesting aspects of writing programs is deciding how to solve a particular problem, especially when multiple solutions exist. For example, there are numerous ways to sort a list of numbers, and each way has its advantages. In order to determine which way is best for a given situation, we need techniques for describing and analyzing solutions formally.

An **algorithm** is a sequence of steps that specifies how to solve a problem. Some algorithms are faster than others, and some use less space in computer memory. **Computer science** is the science of algorithms, including their discovery and analysis. As

you learn to develop algorithms for problems you haven't solved before, you will learn to think like a computer scientist.

Designing algorithms and writing code is difficult and error-prone. For historical reasons, programming errors are called **bugs**, and the process of tracking them down and correcting them is called **debugging**. As you learn to debug your programs, you will develop new problem-solving skills. You will need to think creatively when unexpected errors happen.

Although it can be frustrating, debugging is an intellectually rich, challenging, and interesting part of computer science. In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see. Thinking about how to correct programs and improve their performance sometimes even leads to the discovery of new algorithms.

Debugging Programs

It is a good idea to read this book in front of a computer so you can try out the examples as you go. You can run many of the examples directly in DrJava's Interactions pane (see "[DrJava Interactions](#)" on page 260). But if you put the code in a source file, it will be easier to try out variations.

Whenever you are experimenting with a new feature, you should also try to make mistakes. For example, in the Hello World program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell `println` wrong? These kinds of experiments help you remember what you read. They also help with debugging, because you learn what the error messages mean. It is better to make mistakes now and on purpose than later on and accidentally.

Debugging is like an experimental science: once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one.

Programming and debugging should go hand in hand. Don't just write a bunch of code and then perform trial-and-error debugging until it all works. Instead, start with a program that does *something* and make small modifications, debugging them as you go, until the program does what you want. That way, you will always have a working program, and isolating errors will be easier.

A great example of this principle is the Linux operating system, which contains millions of lines of code. It started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield in *The Linux Users' Guide*, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux."

Finally, programming sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed. Remember that you are not alone, and virtually every programmer has had similar experiences. Don't hesitate to reach out to a friend and ask questions!

Vocabulary

Throughout the book, we try to define each term the first time we use it. At the end of each chapter, we include the new terms and their definitions in order of appearance. If you spend some time learning this vocabulary, you will have an easier time reading the following chapters:

problem solving

The process of formulating a problem, finding a solution, and expressing the solution.

hardware

The electronic and mechanical components of a computer, such as CPUs, RAM, and hard disks.

processo

A computer chip that performs simple instructions like basic arithmetic and logic.

memory

Circuits that store data as long as the computer is turned on. Not to be confused with permanent storage devices like hard disks and flash.

program

A sequence of instructions that specifies how to perform tasks on a computer. Also known as *software*.

programming

The application of problem solving to creating executable computer programs.

statement

Part of a program that specifies one step of an algorithm.

print statement

A statement that causes output to be displayed on the screen.

method

A named sequence of statements.

class

For now, a collection of related methods. (You will see later that there is a lot more to it).

comment

A part of a program that contains information about the program but has no effect when the program runs.

high-level language

A programming language designed to be easy for humans to read and write.

low-level language

A programming language that is designed to be easy for a computer to run. Also called *machine language*.

portable

The ability of a program to run on more than one kind of computer.

interpret

To run a program in a high-level language by translating it one line at a time and immediately executing the corresponding instructions.

compile

To translate a program in a high-level language into a low-level language, all at once, in preparation for later execution.

source code

A program in a high-level language, before being compiled.

object code

The output of the compiler, after translating the program.

executable

Another name for object code that is ready to run on specific hardware.

virtual machine

An emulation of a real machine. The JVM enables a computer to run Java programs.

byte code

A special kind of object code used for Java programs. Byte code is similar to object code, but it is portable like a high-level language.

string

A sequence of characters; the primary data type for text.

newline

A special character signifying the end of a line of text. Also known as *line ending*, *end of line (EOL)*, or *line break*.

escape sequence

A sequence of code that represents a special character when used inside a string.

algorithm

A procedure or formula for solving a problem, with or without a computer.

computer science

The scientific and practical approach to computation and its applications.

bug

An error in a program.

debugging

The process of finding and removing errors.

Exercises

At the end of each chapter, we include exercises you can do with the things you've learned. We encourage you to at least attempt every problem. You can't learn to program only by reading about it; you have to practice.

Before you can compile and run Java programs, you might have to download and install a few tools. There are many good options, but we recommend DrJava, which is an IDE well suited for beginners. Instructions for getting started are in [Appendix A](#).

The code for this chapter is in the *ch01* directory of *ThinkJavaCode2*. See [“Using the Code Examples” on page xii](#) for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

Exercise 1-1.

Computer scientists have the annoying habit of using common English words to mean something other than their common English meanings. For example, in English, statements and comments are the same thing, but in programs they are different.

1. In computer jargon, what's the difference between a *statement* and a *comment*?
2. What does it mean to say that a program is *portable*?
3. In common English, what does the word *compile* mean?
4. What is an *executable*? Why is that word used as a noun?

The Vocabulary section at the end of each chapter is intended to highlight words and phrases that have special meanings in computer science. When you see familiar words, don't assume that you know what they mean!

Exercise 1-2.

Before you do anything else, find out how to compile and run a Java program. Some environments provide sample programs similar to the example in “[The Hello World Program](#)” on page 3.

1. Type in the Hello World program; then compile and run it.
2. Add a print statement that displays a second message after the Hello, World!. Say something witty like, “How are you?” Compile and run the program again.
3. Add a comment to the program (anywhere), recompile, and run it again. The new comment should not affect the result.

This exercise may seem trivial, but it is the starting place for many of the programs we will work with. To debug with confidence, you will need to have confidence in your programming environment.

In some environments, it is easy to lose track of which program is executing. You might find yourself trying to debug one program while you are accidentally running another. Adding (and changing) print statements is a simple way to be sure that the program you are looking at is the program you are running.

Exercise 1-3.

It is a good idea to commit as many errors as you can think of, so that you see what error messages the compiler produces. Sometimes the compiler tells you exactly what is wrong, and all you have to do is fix it. But sometimes the error messages are misleading. Over time you will develop a sense for when you can trust the compiler and when you have to figure things out yourself.

Starting with the Hello World program, try out each of the following errors. After you make each change, compile the program, read the error message (if there is one), and then fix the error.

1. Remove one of the opening curly braces.
2. Remove one of the closing curly braces.
3. Instead of `main`, write `mian`.
4. Remove the word `static`.
5. Remove the word `public`.
6. Remove the word `System`.
7. Replace `println` with **`Println`**.
8. Replace `println` with **`print`**.

9. Delete one parenthesis.
10. Add an extra parenthesis.

Variables and Operators

This chapter describes how to write statements using *variables*, which store values like numbers and words, and *operators*, which are symbols that perform a computation. We also explain three kinds of programming errors and offer additional debugging advice.

To run the examples in this chapter, you will need to create a new Java class with a `main` method (see “[The Hello World Program](#)” on page 3). Throughout the book, we often omit class and method definitions to keep the examples concise.

Declaring Variables

One of the most powerful features of a programming language is the ability to define and manipulate variables. A **variable** is a named location in memory that stores a **value**. Values may be numbers, text, images, sounds, and other types of data. To store a value, you first have to declare a variable:

```
String message;
```

This statement is called a **declaration**, because it declares that the variable `message` has the type `String`. Each variable has a **type** that determines what kind of values it can store. For example, the `int` type can store integers like 1 and -5, and the `char` type can store characters like 'A' and 'z'.

Some types begin with a capital letter and some with lowercase. You will learn the significance of this distinction later, but for now you should take care to get it right. There is no such type as `Int` or `string`.

To declare an integer variable named `x`, you simply type this:

```
int x;
```

Note that `x` is an arbitrary name for the variable. In general, you should use names that indicate what the variables mean:

```
String firstName;  
String lastName;  
int hour, minute;
```

This example declares two variables with type `String` and two with type `int`. The last line shows how to declare multiple variables with the same type: `hour` and `minute` are both integers. Note that each declaration statement ends with a semicolon (`;`).

Variable names usually begin with a lowercase letter, in contrast to class names (like `Hello`) that start with a capital letter. When a variable name contains more than one word (like `firstName`), it is conventional to capitalize the first letter of each subsequent word. Variable names are case-sensitive, so `firstName` is not the same as `first name` or `FirstName`.

You can use any name you want for a variable. But there are about 50 reserved words, called **keywords**, that you are not allowed to use as variable names. These words include `public`, `class`, `static`, `void`, and `int`, which are used by the compiler to analyze the structure of the program.

You can see the [full list of keywords](#), but you don't have to memorize them. Most programming editors provide *syntax highlighting*, which makes different parts of the program appear in different colors. And the compiler will complain even if one does sneak past you and your editor.

Assigning Variables

Now that we have declared some variables, we can use them to store values. We do that with an **assignment** statement:

```
message = "Hello!"; // give message the value "Hello!"  
hour = 11; // assign the value 11 to hour  
minute = 59; // set minute to 59
```

This example shows three assignments, and the comments illustrate different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

- When you declare a variable, you create a named storage location.
- When you make an assignment to a variable, you update its value.

As a general rule, a variable has to have the same type as the value you assign to it. For example, you cannot store a string in `minute` or an integer in `message`. We will show some examples that seem to break this rule, but we'll get to that later.

A common source of confusion is that some strings *look* like integers, but they are not. For example, `message` can contain the string "123", which is made up of the characters '1', '2', and '3'. But that is not the same thing as the integer 123:

```
message = "123";    // legal
message = 123;     // not legal
```

Variables must be **initialized** (assigned for the first time) before they can be used. You can declare a variable and then assign a value later, as in the previous example. You can also declare and initialize on the same line:

```
String message = "Hello!";
int hour = 11;
int minute = 59;
```

Memory Diagrams

Because Java uses the = symbol for assignment, it is tempting to interpret the statement `a = b` as a statement of equality. It is not!

Equality is commutative, and assignment is not. For example, in mathematics if $a = 7$, then $7 = a$. In Java `a = 7;` is a legal assignment statement, but `7 = a;` is not. The left side of an assignment statement has to be a variable name (storage location).

Also, in mathematics, a statement of equality is true for all time. If $a = b$ now, a is always equal to b . In Java, an assignment statement can make two variables equal, but they don't have to stay that way:

```
int a = 5;
int b = a;    // a and b are now equal
a = 3;       // a and b are no longer equal
```

The third line changes the value of `a`, but it does not change the value of `b`, so they are no longer equal.

Taken together, the variables in a program and their current values make up the program's **state**. [Figure 2-1](#) shows the state of the program after these assignment statements run.

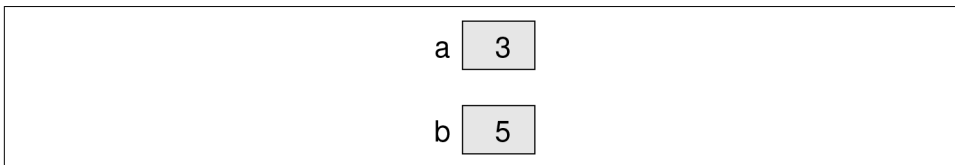


Figure 2-1. Memory diagram of the variables `a` and `b`

Diagrams like this one that show the state of the program are called **memory diagrams**. Each variable is represented with a box showing the name of the variable on the outside and its current value inside.

As the program runs, the state of memory changes, so memory diagrams show only a particular point in time. For example, if we added the line `int c = 0;` to the previous example, the memory diagram would look like [Figure 2-2](#).

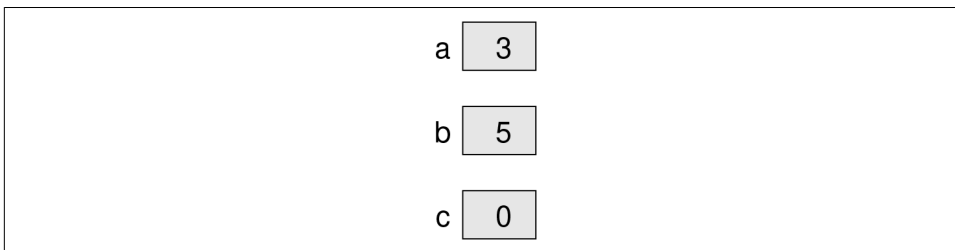


Figure 2-2. Memory diagram of the variables `a`, `b`, and `c`

Printing Variables

You can display the current value of a variable by using `print` or `println`. The following statements declare a variable named `firstLine`, assign it the value "Hello, again!", and display that value:

```
String firstLine = "Hello, again!";  
System.out.println(firstLine);
```

When we talk about displaying a variable, we generally mean the *value* of the variable. To display the *name* of a variable, you have to put it in quotes:

```
System.out.print("The value of firstLine is ");  
System.out.println(firstLine);
```

For this example, the output is as follows:

```
The value of firstLine is Hello, again!
```

Conveniently, the code for displaying a variable is the same regardless of its type. For example:

```
int hour = 11;  
int minute = 59;  
System.out.print("The current time is ");  
System.out.print(hour);  
System.out.print(":");  
System.out.print(minute);  
System.out.println(".");
```

The output of this program is shown here:

```
The current time is 11:59.
```

To output multiple values on the same line, it's common to use several `print` statements followed by `println` at the end. But don't forget the `println`! On many computers, the output from `print` is stored without being displayed until `println` is run; then the entire line is displayed at once. If you omit the `println`, the program might display the stored output at unexpected times or even terminate without displaying anything.

Arithmetic Operators

Operators are symbols that represent simple computations. For example, the addition operator is `+`, subtraction is `-`, multiplication is `*`, and division is `/`.

The following program converts a time of day to minutes:

```
int hour = 11;
int minute = 59;
System.out.print("Number of minutes since midnight: ");
System.out.println(hour * 60 + minute);
```

The output is as follows:

```
Number of minutes since midnight: 719
```

In this program, `hour * 60 + minute` is an **expression**, which represents a single value to be computed (719). When the program runs, each variable is replaced by its current value, and then the operators are applied. The values that operators work with are called **operands**.

Expressions are generally a combination of numbers, variables, and operators. When compiled and executed, they become a single value. For example, the expression `1 + 1` has the value 2. In the expression `hour - 1`, Java replaces the variable with its value, yielding `11 - 1`, which has the value 10.

In the expression `hour * 60 + minute`, both variables get replaced, yielding `11 * 60 + 59`. The multiplication happens first, yielding `660 + 59`. Then the addition yields 719.

Addition, subtraction, and multiplication all do what you expect, but you might be surprised by division. For example, the following fragment tries to compute the fraction of an hour that has elapsed:

```
System.out.print("Fraction of the hour that has passed: ");
System.out.println(minute / 60);
```

The output is as follows:

```
Fraction of the hour that has passed: 0
```

This result often confuses people. The value of `minute` is 59, and 59 divided by 60 should be 0.98333, not 0. The problem is that Java performs *integer division* when the operands are integers. By design, integer division always rounds toward zero, even in cases like this one where the next integer is close.

As an alternative, we can calculate a percentage rather than a fraction:

```
System.out.print("Percent of the hour that has passed: ");
System.out.println(minute * 100 / 60);
```

The new output is as follows:

```
Percent of the hour that has passed: 98
```

Again the result is rounded down, but at least now it's approximately correct.

Floating-Point Numbers

A more general solution is to use **floating-point** numbers, which represent values with decimal places. In Java, the default floating-point type is called `double`, which is short for *double-precision*. You can create `double` variables and assign values to them the same way we did for the other types:

```
double pi;
pi = 3.14159;
```

Java performs *floating-point division* when one or more operands are `double` values. So we can solve the problem from the previous section:

```
double minute = 59.0;
System.out.print("Fraction of the hour that has passed: ");
System.out.println(minute / 60.0);
```

The output is shown here:

```
Fraction of the hour that has passed: 0.9833333333333333
```

Although floating-point numbers are useful, they can be a source of confusion. For example, Java distinguishes the integer value 1 from the floating-point value 1.0, even though they seem to be the same number. They belong to different data types, and strictly speaking, you are not allowed to make assignments between types.

The following is illegal because the variable on the left is an `int` and the value on the right is a `double`:

```
int x = 1.1; // compiler error
```

It is easy to forget this rule, because in many cases Java *automatically* converts from one type to another:

```
double y = 1; // legal, but bad style
```

The preceding example should be illegal, but Java allows it by converting the `int` value `1` to the `double` value `1.0` automatically. This leniency is convenient, but it often causes problems for beginners. For example:

```
double y = 1 / 3; // common mistake
```

You might expect the variable `y` to get the value `0.333333`, which is a legal floating-point value. But instead it gets the value `0.0`. The expression on the right divides two integers, so Java does integer division, which yields the `int` value `0`. Converted to `double`, the value assigned to `y` is `0.0`.

One way to solve this problem (once you figure out the bug) is to make the right-hand side a floating-point expression. The following sets `y` to `0.333333`, as expected:

```
double y = 1.0 / 3.0; // correct
```

As a matter of style, you should always assign floating-point values to floating-point variables. The compiler won't make you do it, but you never know when a simple mistake will come back and haunt you.

Rounding Errors

Most floating-point numbers are only *approximately* correct. Some numbers, like reasonably sized integers, can be represented exactly. But repeating fractions, like $1/3$, and irrational numbers, like π , cannot. To represent these numbers, computers have to round off to the nearest floating-point number.

The difference between the number we want and the floating-point number we get is called **rounding error**. For example, the following two statements should be equivalent:

```
System.out.println(0.1 * 10);
System.out.println(0.1 + 0.1 + 0.1 + 0.1 + 0.1
    + 0.1 + 0.1 + 0.1 + 0.1 + 0.1);
```

But on many machines, the output is as follows:

```
1.0
0.9999999999999999
```

The problem is that 0.1 is a repeating fraction when converted into binary. So its floating-point representation stored in memory is only approximate. When we add up the approximations, the rounding errors accumulate.

For many applications (like computer graphics, encryption, statistical analysis, and multimedia rendering), floating-point arithmetic has benefits that outweigh the costs. But if you need *absolute* precision, use integers instead. For example, consider a bank account with a balance of \$123.45:

```
double balance = 123.45; // potential rounding error
```

In this example, balances will become inaccurate over time as the variable is used in arithmetic operations like deposits and withdrawals. The result would be angry customers and potential lawsuits. You can avoid the problem by representing the balance as an integer:

```
int balance = 12345;    // total number of cents
```

This solution works as long as the number of cents doesn't exceed the largest `int`, which is about 2 billion.

Operators for Strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following expressions are illegal:

```
"Hello" - 1    "World" / 123    "Hello" * "World"
```

The `+` operator works with strings, but it might not do what you expect. For strings, the `+` operator performs **concatenation**, which means joining end-to-end. So `"Hello, " + "World!"` yields the string `"Hello, World!"`.

Likewise if you have a variable called `name` that has type `String`, the expression `"Hello, " + name` appends the value of `name` to the `hello` string, which creates a personalized greeting.

Since addition is defined for both numbers and strings, Java performs automatic conversions you may not expect:

```
System.out.println(1 + 2 + "Hello");  
// the output is 3Hello  
  
System.out.println("Hello" + 1 + 2);  
// the output is Hello12
```

Java executes these operations from left to right. In the first line, `1 + 2` is 3, and `3 + "Hello"` is `"3Hello"`. But in the second line, `"Hello" + 1` is `"Hello1"`, and `"Hello1" + 2` is `"Hello12"`.

When more than one operator appears in an expression, they are evaluated according to the **order of operations**. Generally speaking, Java evaluates operators from left to right (as you saw in the previous section). But for numeric operators, Java follows mathematical conventions:

- Multiplication and division take *precedence* over addition and subtraction, which means they happen first. So `1 + 2 × 3` yields 7, not 9, and `2 + 4 / 2` yields 4, not 3.
- If the operators have the same precedence, they are evaluated from left to right. So in the expression `minute × 100 / 60`, the multiplication happens first; if the

value of `minute` is 59, we get $5900 / 60$, which yields 98. If these same operations had gone from right to left, the result would have been 59×1 , which is incorrect.

- Anytime you want to override the order of operations (or you are not sure what it is), you can use parentheses. Expressions in parentheses are evaluated first, so $(1 + 2) \times 3$ is 9. You can also use parentheses to make an expression easier to read, as in $(\text{minute} \times 100) / 60$, even though it doesn't change the result.

See the official Java tutorials for a complete [table of operator precedence](#). If the order of operations is not obvious when looking at an expression, you can always add parentheses to make it more clear. But over time, you should internalize these kinds of details about the Java language.

Compiler Error Messages

Three kinds of errors can occur in a program: compile-time errors, run-time errors, and logic errors. It is useful to distinguish among them in order to track them down more quickly.

Compile-time errors occur when you violate the rules of the Java language. For example, parentheses and braces have to come in matching pairs. So $(1 + 2)$ is legal, but $8)$ is not. In the latter case, the program cannot be compiled, and the compiler displays a *syntax error*.

Error messages from the compiler usually indicate where in the program the error occurred. Sometimes they can tell you exactly what the error is. As an example, let's get back to the Hello World program from [“The Hello World Program” on page 3](#):

```
public class Hello {  
  
    public static void main(String[] args) {  
        // generate some simple output  
        System.out.println("Hello, World!");  
    }  
}
```

If you forget the semicolon at the end of the print statement, you might get an error message like this:

```
File: Hello.java [line: 5]  
Error: ';' expected
```

That's pretty good: the location of the error is correct, and the error message tells you what's wrong. But error messages are not always easy to understand. Sometimes the compiler reports the place in the program where the error was *detected*, not where it actually occurred. And sometimes the description of the problem is more confusing than helpful.

For example, if you forget the closing brace at the end of `main` (line 6), you might get a message like this:

```
File: Hello.java [line: 7]
Error: reached end of file while parsing
```

There are two problems here. First, the error message is written from the compiler's point of view, not yours. **Parsing** is the process of reading a program before translating; if the compiler gets to the end of the file while still parsing, that means something was omitted. But the compiler doesn't know what. It also doesn't know where. The compiler discovers the error at the end of the program (line 7), but the missing brace should be on the previous line.

Error messages contain useful information, so you should make an effort to read and understand them. But don't take them too literally. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax and other compile-time errors. As you gain experience, you will make fewer mistakes and find them more quickly.

Other Types of Errors

The second type of error is a **run-time error**, so-called because it does not appear until after the program has started running. In Java, these errors occur while the interpreter is executing byte code and something goes wrong. These errors are also called *exceptions* because they usually indicate that something unexpected has happened.

Run-time errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one. When a run-time error occurs, the program “crashes” (terminates) and displays an error message that explains what happened and where. For example, if you accidentally divide by zero, you will get a message like this:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Hello.main(Hello.java:5)
```

Error messages are very useful for debugging. The first line includes the name of the exception, `ArithmeticException`, and a message that indicates more specifically what happened, division by zero. The next line shows the method where the error occurred; `Hello.main` indicates the method `main` in the class `Hello`. It also reports the file where the method is defined, *Hello.java*, and the line number where the error occurred, 5.

The third type of error is a **logic error**. If your program has a logic error, it will compile and run without generating error messages, but it will not do the right thing. Instead, it will do exactly what you told it to do. For example, here is a version of the Hello World program with a logic error:

```
public class Hello {  
  
    public static void main(String[] args) {  
        System.out.println("Hello, ");  
        System.out.println("World!");  
    }  
}
```

This program compiles and runs just fine, but the output is as follows:

```
Hello,  
World!
```

Assuming that we wanted the output on one line, this is not correct. The problem is that the first line uses `println`, when we probably meant to use `print` (see the “good-bye, cruel world” example of [“Displaying Two Messages” on page 6](#)).

Identifying logic errors can be hard because you have to work backward, looking at the output of the program, trying to figure out why it is doing the wrong thing, and how to make it do the right thing. Usually, the compiler and the interpreter can’t help you, since they don’t know what the right thing is.

Vocabulary

variable

A named storage location for values. All variables have a type, which is declared when the variable is created.

value

A number, string, or other data that can be stored in a variable. Every value belongs to a type (e.g., `int` or `String`).

declaration

A statement that creates a new variable and specifies its type.

type

Mathematically speaking, a set of values. The type of a variable determines which values it can have.

keyword

A reserved word used by the compiler to analyze programs. You cannot use keywords (like `public`, `class`, and `void`) as variable names.

assignment

A statement that gives a value to a variable.

initialize

To assign a variable for the first time.

state

The variables in a program and their current values.

memory diagram

A graphical representation of the state of a program at a point in time.

operator

A symbol that represents a computation like addition, multiplication, or string concatenation.

expression

A combination of variables, operators, and values that represents a single value. Expressions also have types, as determined by their operators and operands.

operand

One of the values on which an operator operates. Most operators in Java require two operands.

floating-point

A data type that represents numbers with an integer part and a fractional part. In Java, the default floating-point type is `double`.

rounding error

The difference between the number we want to represent and the nearest floating-point number.

concatenate

To join two values, often strings, end to end.

order of operations

The rules that determine in what order expressions are evaluated. Also known as *operator precedence*.

compile-time error

An error in the source code that makes it impossible to compile. Also called a *syntax error*.

parse

To analyze the structure of a program; what the compiler does first.

run-time error

An error in a program that makes it impossible to run to completion. Also called an *exception*.

logic error

An error in a program that makes it do something other than what the programmer intended.

Exercises

The code for this chapter is in the *ch02* directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

If you have not already read “DrJava Interactions” on page 260, now might be a good time. It describes the DrJava Interactions pane, which is a useful way to develop and test short fragments of code without writing a complete class definition.

Exercise 2-1.

If you are using this book in a class, you might enjoy this exercise. Find a partner and play Stump the Chump.

Start with a program that compiles and runs correctly. One player looks away, while the other player adds an error to the program. Then the first player tries to find and fix the error. You get two points if you find the error without compiling the program, one point if you find it using the compiler, and your opponent gets a point if you don't find it.

Exercise 2-2.

The point of this exercise is (1) to use string concatenation to display values with different types (`int` and `String`), and (2) to practice developing programs gradually by adding a few statements at a time.

1. Create a new program named `Date.java`. Copy or type in something like the Hello World program and make sure you can compile and run it.
2. Following the example in “Printing Variables” on page 18, write a program that creates variables named `day`, `date`, `month`, and `year`. The variable `day` will contain the day of the week (like Friday), and `date` will contain the day of the month (like the 13th). Assign values to those variables that represent today's date.
3. Display the value of each variable on a line by itself. This is an intermediate step that is useful for checking that everything is working so far. Compile and run your program before moving on.
4. Modify the program so that it displays the date in standard American format; for example: Thursday, July 18, 2019.
5. Modify the program so it also displays the date in European format. The final output should be as follows:

```
American format: Thursday, July 18, 2019
European format: Thursday 18 July 2019
```

Exercise 2-3.

The point of this exercise is to (1) use some of the arithmetic operators, and (2) start thinking about compound entities (like time of day) that are represented with multiple values.

1. Create a new program called `Time.java`. From now on, we won't remind you to start with a small, working program, but you should.
2. Following the example program in “[Printing Variables](#)” on page 18, create variables named `hour`, `minute`, and `second`. Assign values that are roughly the current time. Use a 24-hour clock so that at 2 p.m. the value of `hour` is 14.
3. Make the program calculate and display the number of seconds since midnight.
4. Calculate and display the number of seconds remaining in the day.
5. Calculate and display the percentage of the day that has passed. You might run into problems when computing percentages with integers, so consider using floating-point.
6. Change the values of `hour`, `minute`, and `second` to reflect the current time. Then write code to compute the elapsed time since you started working on this exercise.

Hint: You might want to use additional variables to hold values during the computation. Variables that are used in a computation but never displayed are sometimes called *intermediate* or *temporary* variables.

Input and Output

The programs you've looked at so far simply display messages, which doesn't really involve that much computation. This chapter shows you how to read input from the keyboard, use that input to calculate a result, and then format that result for output.

The System Class

We have been using `System.out.println` for a while, but you might not have thought about what it means. `System` is a class that provides methods related to the *system*, or environment, where programs run. It also provides `System.out`, which is a special value that has additional methods (like `println`) for displaying output.

In fact, we can use `System.out.println` to display the value of `System.out`:

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

The result is shown here:

```
java.io.PrintStream@685d72cd
```

This output indicates that `System.out` is a `PrintStream`, which is defined in a package called `java.io`. A **package** is a collection of related classes; `java.io` contains classes for *I/O* which stands for *input and output*.

The numbers and letters after the `@` sign are the **address** of `System.out`, represented as a hexadecimal (base 16) number. The address of a value is its location in the computer's memory, which might be different on different computers. In this example, the address is `685d72cd`, but if you run the same code, you will likely get something else.

As shown in [Figure 3-1](#), `System` is defined in a file called `System.java`, and `PrintStream` is defined in `PrintStream.java`. These files are part of the Java **library**, which is

an extensive collection of classes that you can use in your programs. The source code for these classes is usually included with the compiler (see “Java Library Source” on page 152).

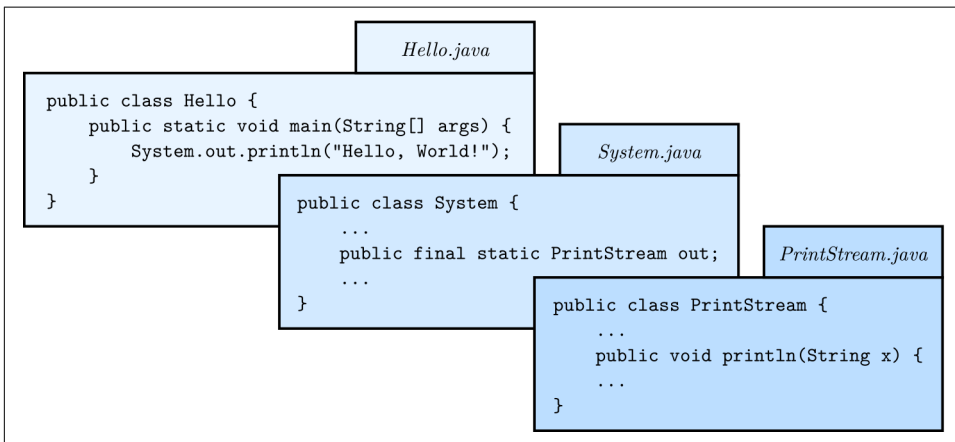


Figure 3-1. `System.out.println` refers to the `out` variable of the `System` class, which is a `PrintStream` that provides a method called `println`

The Scanner Class

The `System` class also provides the special value `System.in`, which is an `InputStream` that has methods for reading input from the keyboard. These methods are not convenient to use, but, fortunately, Java provides other classes that make it easy to handle common input tasks.

For example, `Scanner` is a class that provides methods for inputting words, numbers, and other data. `Scanner` is provided by `java.util`, which is a package that contains various *utility classes*. Before you can use `Scanner`, you have to import it like this:

```
import java.util.Scanner;
```

This **import statement** tells the compiler that when you refer to `Scanner`, you mean the one defined in `java.util`. Using an import statement is necessary because there might be another class named `Scanner` in another package.

Next you have to initialize the `Scanner`. This line declares a `Scanner` variable named `in` and creates a `Scanner` that reads input from `System.in`:

```
Scanner in = new Scanner(System.in);
```

The `Scanner` class provides a method called `nextLine` that reads a line of input from the keyboard and returns a `String`. Here’s a complete example that reads two lines and repeats them back to the user:


```

import java.util.Scanner;

public class Echo {

    public static void main(String[] args) {
        String line;
        Scanner in = new Scanner(System.in);

        System.out.print("Type something: ");
        line = in.nextLine();
        System.out.println("You said: " + line);

        System.out.print("Type something else: ");
        line = in.nextLine();
        System.out.println("You also said: " + line);
    }
}

```

Import statements can't be inside a class definition. By convention, they are usually at the beginning of the file. If you omit the import statement, you get a compiler error like *cannot find symbol*. That means the compiler doesn't know where to find the definition for `Scanner`.

You might wonder why we can use the `System` class without importing it. `System` belongs to the `java.lang` package, which is imported automatically. According to the documentation, `java.lang` “provides classes that are fundamental to the design of the Java programming language.” The `String` class is also part of `java.lang`.

Language Elements

At this point, we have seen nearly all of the organizational units that make up Java programs. [Figure 3-2](#) shows how these *language elements* are related.

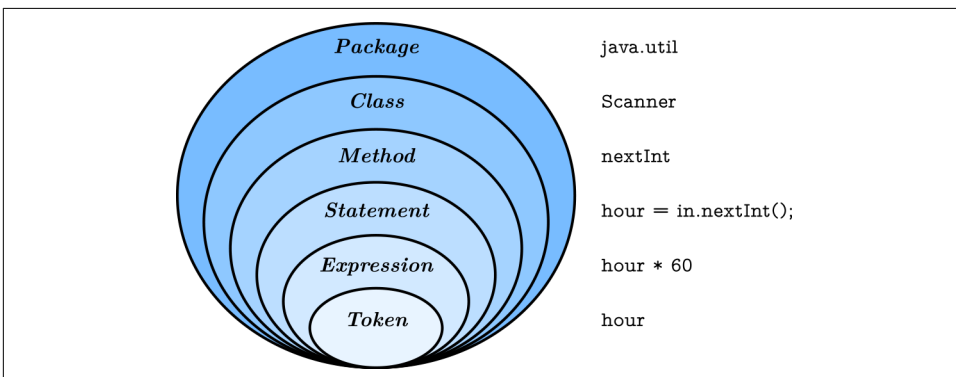


Figure 3-2. Elements of the Java language, from largest to smallest

Java applications are typically organized into packages (like `java.io` and `java.util`) that include multiple classes (like `PrintStream` and `Scanner`). Each class defines its own methods (like `println` and `nextLine`), and each method is a sequence of statements.

Each statement performs one or more computations, depending on how many expressions it has, and each expression represents a single value to compute. For example, the assignment statement `hours = minutes / 60.0;` contains a single expression: `minutes / 60.0`.

Tokens are the most basic elements of a program, including numbers, variable names, operators, keywords, parentheses, braces, and semicolons. In the previous example, the tokens are `hours`, `=`, `minutes`, `/`, `60.0`, and `;` (spaces are ignored by the compiler).

Knowing this terminology is helpful, because error messages often say things like `not a statement` or `illegal start of expression` or `unexpected token`. Comparing Java to English, statements are complete sentences, expressions are phrases, and tokens are individual words and punctuation marks.

Note there is a big difference between the Java *language*, which defines the elements in [Figure 3-2](#), and the Java *library*, which provides the built-in classes that you can import. For example, the keywords `public` and `class` are part of the Java language, but the names `PrintStream` and `Scanner` are not.

The standard edition of Java comes with *several thousand* classes you can use, which can be both exciting and intimidating. You can browse [this library](#). Interestingly, most of the Java library is written in Java.

Literals and Constants

Although most of the world has adopted the metric system for weights and measures, some countries are stuck with imperial units. For example, when talking with friends in Europe about the weather, people in the United States might have to convert from Celsius to Fahrenheit and back. Or they might want to convert height in inches to centimeters.

We can write a program to help. We'll use a `Scanner` to input a measurement in inches, convert to centimeters, and then display the results. The following lines declare the variables and create the `Scanner`:

```
int inch;  
double cm;  
Scanner in = new Scanner(System.in);
```

The next step is to prompt the user for the input. We'll use `print` instead of `println` so the user can enter the input on the same line as the **prompt**. And we'll use the `Scanner` method `nextInt`, which reads input from the keyboard and converts it to an integer:

```
System.out.print("How many inches? ");
inch = in.nextInt();
```

Next we multiply the number of inches by 2.54, since that's how many centimeters there are per inch, and display the results:

```
cm = inch * 2.54;
System.out.print(inch + " in = ");
System.out.println(cm + " cm");
```

This code works correctly, but it has a minor problem. If another programmer reads this code, they might wonder where 2.54 comes from. For the benefit of others (and yourself in the future), it would be better to assign this value to a variable with a meaningful name.

A value that appears in a program, like the number 2.54, is called a **literal**. In general, there's nothing wrong with literals. But when numbers like 2.54 appear in an expression with no explanation, they make the code hard to read. And if the same value appears many times and could change in the future, it makes the code hard to maintain.

Values like 2.54 are sometimes called **magic numbers** (with the implication that being magic is not a good thing). A good practice is to assign magic numbers to variables with meaningful names, like this:

```
double cmPerInch = 2.54;
cm = inch * cmPerInch;
```

This version is easier to read and less error-prone, but it still has a problem. Variables can vary (hence the term), but the number of centimeters in an inch does not. Once we assign a value to `cmPerInch`, it should never change. Java provides the keyword `final`, a language feature that enforces this rule:

```
final double CM_PER_INCH = 2.54;
```

Declaring that a variable is `final` means that it cannot be reassigned once it has been initialized. If you try, the compiler gives an error.

Variables declared as `final` are called **constants**. By convention, names for constants are all uppercase, with the underscore character (`_`) between words.

Formatting Output

When you output a double by using `print` or `println`, it displays up to 16 decimal places:

```
System.out.print(4.0 / 3.0);
```

The result is as follows:

```
1.3333333333333333
```

That might be more than you want. `System.out` provides another method, called `printf`, that gives you more control of the format. The *f* in `printf` stands for *formatted*. Here's an example:

```
System.out.printf("Four thirds = %.3f", 4.0 / 3.0);
```

The first value in the parentheses is a **format string** that specifies how the output should be displayed. This format string contains ordinary text followed by a **format specifier**, which is a special sequence that starts with a percent sign. The format specifier `%.3f` indicates that the following value should be displayed as floating-point, rounded to three decimal places. The result is shown here:

```
Four thirds = 1.333
```

The format string can contain any number of format specifiers; here's an example with two of them:

```
int inch = 100;  
double cm = inch * CM_PER_INCH;  
System.out.printf("%d in = %f cm\n", inch, cm);
```

The result is as follows:

```
100 in = 254.000000 cm
```

Like `print`, `printf` does not append a newline. So format strings often end with a newline character.

The format specifier `%d` displays integer values (*d* stands for *decimal*, meaning base 10 integer). The values are matched up with the format specifiers in order, so `inch` is displayed using `%d`, and `cm` is displayed using `%f`.

Learning about format strings is like learning a sublanguage within Java. There are many options, and the details can be overwhelming. [Table 3-1](#) lists a few common uses, to give you an idea of how things work.

Table 3-1. Example format specifiers

%d	Integer in base 10 (decimal)	12345
%,d	Integer with comma separators	12,345
%08d	Padded with zeros, at least eight digits wide	00012345
%f	Floating-point number	6.789000
%.2f	Rounded to two decimal places	6.79
%s	String of characters	"Hello"
%x	Integer in base 16 (hexadecimal)	bc614e

For more details, refer to the documentation of `java.util.Formatter`. The easiest way to find documentation for Java classes is to do a web search for “Java” and the name of the class.

Reading Error Messages

Notice that the values you pass to `printf` are separated by commas. If you are used to using the `+` operator to concatenate strings, you might write something like this by accident:

```
System.out.printf("inches = %d" + inch); // error
```

This line of code is legal, so the compiler won't catch the mistake. Instead, when you run the program, it causes an exception:

```
Exception in thread "main" java.util.MissingFormatArgumentException:
Format specifier '%d'
    at java.util.Formatter.format(Formatter.java:2519)
    at java.io.PrintStream.format(PrintStream.java:970)
    at java.io.PrintStream.printf(PrintStream.java:871)
    at Example.main(Example.java:10)
```

As you saw in “Other Types of Errors” on page 24, the error message includes the name of the exception, `MissingFormatArgumentException`, followed by additional details, `Format specifier '%d'`. That means it doesn't know what value to substitute for `%d`.

The problem is that concatenation happens first, before `printf` executes. If the value of `inch` is `100`, the result of concatenation is `"inches = %d100"`. So `printf` gets the format string, but it doesn't get any values to format.

The error message also includes a **stack trace** that shows the method that was running when the error was detected, `java.util.Formatter.format`; the method that

ran it, `java.io.PrintStream.format`; the method that ran *that*, `java.io.PrintStream.printf`; and finally the method you actually wrote, `Example.main`.

Each line also names the source file of the method and the line it was on (e.g., `Example.java:10`). That's a lot of information, and it includes method names and file-names you have no reason to know at this point. But don't be overwhelmed.

When you see an error message like this, read the first line carefully to see *what* happened. Then read the last line to see *where* it happened. In some IDEs, you can click the error message, and it will take you to the line of code that was running. But remember that where the error is discovered is not always where it was caused.

Type Cast Operators

Now suppose we have a measurement in centimeters, and we want to round it off to the nearest inch. It is tempting to write this:

```
inch = cm / CM_PER_INCH; // syntax error
```

But the result is an error—you get something like `incompatible types: possible lossy conversion from double to int`. The problem is that the value on the right is floating-point, and the variable on the left is an integer.

Java converts an `int` to a `double` automatically, since no information is lost in the process. On the other hand, going from `double` to `int` would lose the decimal places. Java doesn't perform this operation automatically in order to ensure that you are aware of the loss of the fractional part of the number.

The simplest way to convert a floating-point value to an integer is to use a **type cast**, so called because it molds, or *casts*, a value from one type to another. The syntax for type casting is to put the name of the type in parentheses and use it as an operator:

```
double pi = 3.14159;  
int x = (int) pi;
```

The `(int)` operator has the effect of converting what follows into an integer. In this example, `x` gets the value 3. Like integer division, casting to an integer always rounds toward zero, even if the fractional part is `0.999999` (or `-0.999999`). In other words, it simply throws away the fractional part.

In order to use a cast operator, the types must be compatible. For example, you can't cast a `String` to an `int` because a string is not a number:

```
String str = "3";  
int x = (int) str; // error: incompatible types
```

Type casting takes precedence over arithmetic operations. In the following example, the value of `pi` gets converted to an integer before the multiplication:

```
double pi = 3.14159;
double x = (int) pi * 20.0; // result is 60.0, not 62.0
```

Keeping that in mind, here's how we can convert centimeters to inches:

```
inch = (int) (cm / CM_PER_INCH);
System.out.printf("%f cm = %d in\n", cent, inch);
```

The parentheses after the cast operator require the division to happen before the type cast. And the result is rounded toward zero. You will see in the next chapter how to round floating-point numbers to the closest integer.

Remainder Operator

Let's take the example one step further: suppose you have a measurement in inches and you want to convert to feet and inches. The goal is divide by 12 (the number of inches in a foot) and keep the remainder.

You have already seen the division operation (`/`), which computes the quotient of two numbers. If the numbers are integers, the operation performs integer division. Java also provides the **modulo** operation (`%`), which divides two numbers and computes the remainder.

Using division and modulo, we can convert to feet and inches like this:

```
feet = 76 / 12; // quotient
inches = 76 % 12; // remainder
```

The first line yields 6. The second line, which is pronounced “76 mod 12”, yields 4. So 76 inches is 6 feet, 4 inches.

Many people (and textbooks) incorrectly refer to `%` as the *modulus operator*. In mathematics, however, **modulus** is the number you're dividing by. In the previous example, the modulus is 12.

The Java language specification refers to `%` as the *remainder operator*. The remainder operator looks like a percent sign, but you might find it helpful to think of it as a division sign (\div) rotated to the left.

Modular arithmetic turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if `x % y` is 0, then `x` is divisible by `y`. You can use the remainder operator to “extract” digits from a number: `x % 10` yields the rightmost digit of `x`, and `x % 100` yields the last two digits. And many encryption algorithms use the remainder operator extensively.

Putting It All Together

At this point, you have seen enough Java to write useful programs that solve everyday problems. You can (1) import Java library classes, (2) create a `Scanner`, (3) get input from the keyboard, (4) format output with `printf`, and (5) divide and mod integers. Now we will put everything together in a complete program:

```
import java.util.Scanner;

/**
 * Converts centimeters to feet and inches.
 */
public class Convert {

    public static void main(String[] args) {
        double cm;
        int feet, inches, remainder;
        final double CM_PER_INCH = 2.54;
        final int IN_PER_FOOT = 12;
        Scanner in = new Scanner(System.in);

        // prompt the user and get the value
        System.out.print("Exactly how many cm? ");
        cm = in.nextDouble();

        // convert and output the result
        inches = (int) (cm / CM_PER_INCH);
        feet = inches / IN_PER_FOOT;
        remainder = inches % IN_PER_FOOT;
        System.out.printf("%.2f cm = %d ft, %d in\n",
                           cm, feet, remainder);
    }
}
```

Although not required, all variables and constants are declared at the top of `main`. This practice makes it easier to find their types later on, and it helps the reader know what data is involved in the algorithm.

For readability, each major step of the algorithm is separated by a blank line and begins with a comment. The class also includes a documentation comment (`/**`), which you can learn more about in [Appendix B](#).

Many algorithms, including the `Convert` program, perform division and modulo together. In both steps, you divide by the same number (`IN_PER_FOOT`).

When statements including `System.out.printf` get long (generally wider than 80 characters), a common style convention is to break them across multiple lines. The reader should never have to scroll horizontally.

The Scanner Bug

Now that you've had some experience with Scanner, we want to warn you about an unexpected behavior. The following code fragment asks users for their name and age:

```
System.out.print("What is your name? ");
name = in.nextLine();
System.out.print("What is your age? ");
age = in.nextInt();
System.out.printf("Hello %s, age %d\n", name, age);
```

The output might look something like this:

```
Hello Grace Hopper, age 45
```

When you read a String followed by an int, everything works just fine. But when you read an int followed by a String, something strange happens:

```
System.out.print("What is your age? ");
age = in.nextInt();
System.out.print("What is your name? ");
name = in.nextLine();
System.out.printf("Hello %s, age %d\n", name, age);
```

Try running this example code. It doesn't let you input your name, and it immediately displays the output:

```
What is your name? Hello , age 45
```

To understand what is happening, you need to realize that Scanner doesn't see input as multiple lines as we do. Instead, it gets a *stream of characters*, as shown in [Figure 3-3](#).

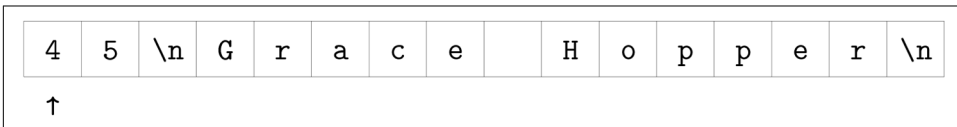


Figure 3-3. A stream of characters as seen by a Scanner

The arrow indicates the next character to be read by Scanner. When you run `nextInt`, it reads characters until it gets to a nondigit. [Figure 3-4](#) shows the state of the stream after `nextInt` runs.

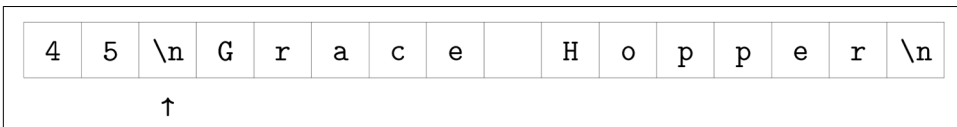


Figure 3-4. A stream of characters after `nextInt` runs

At this point, `nextInt` returns the value 45. The program then displays the prompt "What is your name? " and runs `nextLine`, which reads characters until it gets to a newline. But since the next character is already a newline, `nextLine` returns the empty string "".

To solve this problem, you need an extra `nextLine` after `nextInt`:

```
System.out.print("What is your age? ");
age = in.nextInt();
in.nextLine(); // read the newline
System.out.print("What is your name? ");
name = in.nextLine();
System.out.printf("Hello %s, age %d\n", name, age);
```

This technique is common when reading `int` or `double` values that appear on their own line. First you read the number, and then you read the rest of the line, which is just a newline character.

Vocabulary

package

A directory of classes that are related to each other.

address

The location of a value in computer memory, often represented as a hexadecimal integer.

library

A collection of packages and classes that are available for use in other programs.

import statement

A statement that allows programs to use classes defined in other packages.

token

The smallest unit of source code, such as an individual word, literal value, or symbol.

prompt

A brief message displayed in a print statement that asks the user for input.

literal

A value that appears in source code. For example, "Hello" is a string literal, and 74 is an integer literal.

magic number

A number that appears without explanation as part of an expression. It should generally be replaced with a constant.

constant

A variable, declared as `final`, whose value cannot be changed.

format string

The string in `System.out.printf` that specifies the format of the output.

format specifier

A special code that begins with a percent sign and specifies the data type and format of the corresponding value.

stack trace

An error message that shows the methods that were running when an exception occurs.

type cast

An operation that explicitly converts one data type into another. In Java, it appears as a type name in parentheses, like `(int)`.

modulo

An operation that yields the remainder when one integer is divided by another. In Java, it is denoted with a percent sign: `5 % 2` is 1.

modulus

The value of `b` in the expression `a % b`. It often represents unit conversions, such as 24 hours in a day, 60 minutes in an hour, etc.

Exercises

The code for this chapter is in the `ch03` directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

If you have not already read “Command-Line Interface” on page 261, now might be a good time. It describes the command-line interface, which is a powerful and efficient way to interact with your computer.

Exercise 3-1.

When you use `printf`, the Java compiler does not check your format string. See what happens if you try to display a value with type `int` using `%f`. And what happens if you display a `double` using `%d`? What if you use two format specifiers, but then provide only one value?

Exercise 3-2.

Write a program that converts a temperature from Celsius to Fahrenheit. It should (1) prompt the user for input, (2) read a double value from the keyboard, (3) calculate the result, and (4) format the output to one decimal place.

When it's finished, it should work like this:

```
Enter a temperature in Celsius: 24
24.0 C = 75.2 F
```

Here is the formula to do the conversion:

$$F = C \times \frac{9}{5} + 32$$

Hint: Be careful not to use integer division!

Exercise 3-3.

Write a program that converts a total number of seconds to hours, minutes, and seconds. It should (1) prompt the user for input, (2) read an integer from the keyboard, (3) calculate the result, and (4) use `printf` to display the output. For example, "5000 seconds = 1 hours, 23 minutes, and 20 seconds".

Hint: Use the remainder operator.

Exercise 3-4.

The goal of this exercise is to program a Guess My Number game. When it's finished, it should work like this:

```
I'm thinking of a number between 1 and 100
(including both). Can you guess what it is?
Type a number: 45
Your guess is: 45
The number I was thinking of is: 14
You were off by: 31
```

To choose a random number, you can use the `Random` class in `java.util`. Here's how it works:

```
import java.util.Random;

public class GuessStarter {

    public static void main(String[] args) {
        // pick a random number
        Random random = new Random();
        int number = random.nextInt(100) + 1;
        System.out.println(number);
    }
}
```

Like the `Scanner` class in this chapter, `Random` has to be imported before we can use it. And as with `Scanner`, we have to use the `new` operator to create a `Random` (number generator).

Then we can use the method `nextInt` to generate a random number. In this example, the result of `nextInt(100)` will be between 0 and 99, including both. Adding 1 yields a number between 1 and 100, including both.

1. The definition of `GuessStarter` is in a file called *GuessStarter.java*, in the directory called *ch03*, in the repository for this book.
2. Compile and run this program.
3. Modify the program to prompt the user; then use a `Scanner` to read a line of user input. Compile and test the program.
4. Read the user input as an integer and display the result. Again, compile and test.
5. Compute and display the difference between the user's guess and the number that was generated.

Methods and Testing

So far, we've written programs that have only one method, named `main`. In this chapter, we'll show you how to organize programs into multiple methods. We'll also take a look at the `Math` class, which provides methods for common mathematical operations. Finally, we'll discuss strategies for incrementally developing and testing your code.

Defining New Methods

Some methods perform a computation and return a result. For example, `nextDouble` reads input from the keyboard and returns it as a `double`. Other methods, like `println`, carry out a sequence of actions without returning a result. Java uses the keyword `void` to define such methods:

```
public static void newLine() {
    System.out.println();
}

public static void main(String[] args) {
    System.out.println("First line.");
    newLine();
    System.out.println("Second line.");
}
```

In this example, the `newLine` and `main` methods are both `public`, which means they can be **invoked** (or *called*) from other classes. And they are both `void`, which means that they don't return a result (in contrast to `nextDouble`). The output of the program is shown here:

```
First line.

Second line.
```

Notice the extra space between the lines. If we wanted more space between them, we could invoke the same method repeatedly. Or we could write yet another method (named `threeLine`) that displays three blank lines:

```
public class NewLine {  
  
    public static void newLine() {  
        System.out.println();  
    }  
  
    public static void threeLine() {  
        newLine();  
        newLine();  
        newLine();  
    }  
  
    public static void main(String[] args) {  
        System.out.println("First line.");  
        threeLine();  
        System.out.println("Second line.");  
    }  
}
```

In this example, the name of the class is `NewLine`. By convention, class names begin with a capital letter. `NewLine` contains three methods, `newLine`, `threeLine`, and `main`. Remember that Java is case-sensitive, so `NewLine` and `newLine` are not the same.

By convention, method names begin with a lowercase letter and use *camel case*, which is a cute name for `jammingWordsTogetherLikeThis`. You can use any name you want for methods, except `main` or any of the Java keywords.

Flow of Execution

When you look at a class definition that contains several methods, it is tempting to read it from top to bottom. But that is *not* the **flow of execution**, or the order the program actually runs. The `NewLine` program runs methods in the opposite order than they are listed.

Programs always begin at the first statement of `main`, regardless of where it is in the source file. Statements are executed one at a time, in order, until you reach a method invocation, which you can think of as a detour. Instead of going to the next statement, you jump to the first line of the invoked method, execute all the statements there, and then come back and pick up exactly where you left off.

That sounds simple enough, but remember that one method can invoke another one. In the middle of `main`, the previous example goes off to execute the statements in `threeLine`. While in `threeLine`, it goes off to execute `newLine`. Then `newLine` invokes `println`, which causes yet another detour.

Fortunately, Java is good at keeping track of which methods are running. So when `println` completes, it picks up where it left off in `newline`; when `newline` completes, it goes back to `threeline`; and when `threeline` completes, it gets back to `main`.

Beginners often wonder why it's worth the trouble to write other methods, when they could just do everything in `main`. The `NewLine` example demonstrates a few reasons:

- Creating a new method allows you to *name a block of statements*, which makes the code easier to read and understand.
- Introducing new methods can *make the program shorter* by eliminating repetitive code. For example, to display nine consecutive newlines, you could invoke `threeLine` three times.
- A common problem-solving technique is to *break problems down* into subproblems. Methods allow you to focus on each subproblem in isolation, and then compose them into a complete solution.

Perhaps most importantly, organizing your code into multiple methods allows you to test individual parts of your program separately. It's easier to get a complex program working if you know that each method works correctly.

Parameters and Arguments

Some of the methods we have used require **arguments**, which are the values you provide in parentheses when you invoke the method.

For example, the `println` method takes a `String` argument. To display a message, you have to provide the message: `System.out.println("Hello")`. Similarly, the `printf` method can take multiple arguments. The statement `System.out.printf("%d in = %f cm", inch, cm)` has three arguments: the format string, the `inch` value, and the `cm` value.

When you invoke a method, you provide the arguments. When you define a method, you name the **parameters**, which are variables that indicate what arguments are required. The following class shows an example:

```
public class PrintTwice {  
  
    public static void printTwice(String s) {  
        System.out.println(s);  
        System.out.println(s);  
    }  
  
    public static void main(String[] args) {  
        printTwice("Don't make me say this twice!");  
    }  
}
```

The `printTwice` method has a parameter named `s` with type `String`. When you invoke `printTwice`, you have to provide an argument with type `String`.

Before the method executes, the argument gets assigned to the parameter. In the `printTwice` example, the argument "Don't make me say this twice!" gets assigned to the parameter `s`.

This process is called **parameter passing**, because the value gets passed from outside the method to the inside. An argument can be any kind of expression, so if you have a `String` variable, you can use its value as an argument:

```
String message = "Never say never.";
printTwice(message);
```

The value you provide as an argument must have the same (or compatible) type as the parameter. For example, if you try this:

```
printTwice(17); // syntax error
```

You will get an error message like this:

```
File: Test.java [line: 10]
Error: method printTwice in class Test cannot be applied
      to given types;
      required: java.lang.String
      found: int
      reason: actual argument int cannot be converted to
              java.lang.String by method invocation conversion
```

This error message says that it found an `int` argument, but the required parameter should be a `String`. In the case of `printTwice`, Java won't convert the integer 17 to the string "17" automatically.

Sometimes Java can convert an argument from one type to another automatically. For example, `Math.sqrt` requires a `double`, but if you invoke `Math.sqrt(25)`, the integer value 25 is automatically converted to the floating-point value 25.0.

Parameters and other variables exist only inside their own methods. In the `printTwice` example, there is no such thing as `s` in the `main` method. If you try to use it there, you'll get a compiler error.

Similarly, inside `printTwice` there is no such thing as `message`. That variable belongs to `main`. Because variables exist only inside the methods where they are defined, they are often called **local variables**.

Multiple Parameters

Here is an example of a method that takes two parameters:

```
public static void printTime(int hour, int minute) {
    System.out.print(hour);
    System.out.print(":");
    System.out.println(minute);
}
```

To invoke this method, we have to provide two integers as arguments:

```
int hour = 11;
int minute = 59;
printTime(hour, minute);
```

Beginners sometimes make the mistake of *declaring* the arguments:

```
int hour = 11;
int minute = 59;
printTime(int hour, int minute); // syntax error
```

That's a syntax error, because the compiler sees `int hour` and `int minute` as variable declarations, not expressions that represent values. You wouldn't declare the types of the arguments if they were simply integers:

```
printTime(int 11, int 59); // syntax error
```

Pulling together the code fragments, here is the complete program:

```
public class PrintTime {

    public static void printTime(int hour, int minute) {
        System.out.print(hour);
        System.out.print(":");
        System.out.println(minute);
    }

    public static void main(String[] args) {
        int hour = 11;
        int minute = 59;
        printTime(hour, minute);
    }
}
```

`printTime` has two parameters, named `hour` and `minute`. And `main` has two variables, also named `hour` and `minute`. Although they have the same names, these variables are *not* the same. The `hour` in `printTime` and the `hour` in `main` refer to different memory locations, and they can have different values.

For example, you could invoke `printTime` like this:

```
int hour = 11;
int minute = 59;
printTime(hour + 1, 0);
```

Before the method is invoked, Java evaluates the arguments; in this example, the results are 12 and 0. Then it assigns those values to the parameters. Inside `printTime`, the value of `hour` is 12, not 11, and the value of `minute` is 0, not 59. Furthermore, if `printTime` modifies one of its parameters, that change has no effect on the variables in `main`.

Stack Diagrams

One way to keep track of variables is to draw a **stack diagram**, which is a memory diagram (see “[Memory Diagrams](#)” on page 17) that shows currently running methods. For each method, there is a box called a **frame** that contains the method’s parameters and local variables. The name of the method appears outside the frame; the variables and parameters appear inside.

As with memory diagrams, stack diagrams show variables and methods at a particular point in time. [Figure 4-1](#) is a stack diagram at the beginning of the `printTime` method. Notice that `main` is on top, because it executed first.

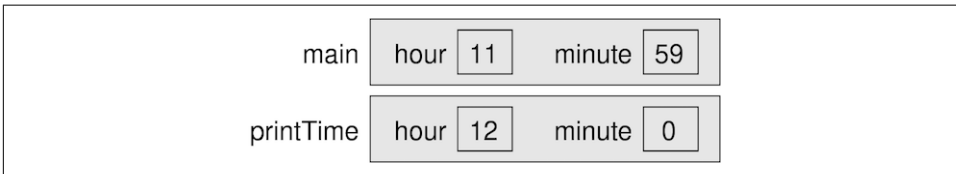


Figure 4-1. Stack diagram for `printTime(hour + 1, 0)`.

Stack diagrams help you to visualize the **scope** of a variable, which is the area of a program where a variable can be used.

Stack diagrams are a good mental model for how variables and methods work at runtime. Learning to trace the execution of a program on paper (or on a whiteboard) is a useful skill for communicating with other programmers.

Educational tools can automatically draw stack diagrams for you. For example, [Java Tutor](#) allows you to step through an entire program, both forward and backward, and see the stack frames and variables at each step. If you haven’t already, you should check out the Java examples on that website.

Math Methods

You don't always have to write new methods to get work done. As a reminder, the Java library contains thousands of classes you can use. For example, the `Math` class provides common mathematical operations:

```
double root = Math.sqrt(17.0);
double angle = 1.5;
double height = Math.sin(angle);
```

The first line sets `root` to the square root of 17. The third line finds the sine of 1.5 (the value of `angle`). `Math` is in the `java.lang` package, so you don't have to import it.

Values for the trigonometric functions—`sin`, `cos`, and `tan`—must be in *radians*. To convert from degrees to radians, you divide by 180 and multiply by π . Conveniently, the `Math` class provides a constant named `PI` that contains an approximation of π :

```
double degrees = 90;
double angle = degrees / 180.0 * Math.PI;
```

Notice that `PI` is in capital letters. Java does not recognize `pi`, `π`, or `pie`. Also, `PI` is the name of a constant, not a method, so it doesn't have parentheses. The same is true for the constant `Math.E`, which approximates Euler's number.

Converting to and from radians is a common operation, so the `Math` class provides methods that do that for you:

```
double radians = Math.toRadians(180.0);
double degrees = Math.toDegrees(Math.PI);
```

Another useful method is `round`, which rounds a floating-point value to the nearest integer and returns a `long`. The following result is 63 (rounded up from 62.8319):

```
long x = Math.round(Math.PI * 20.0);
```

A `long` is like an `int`, but bigger. More specifically, an `int` uses 32 bits of memory; the largest value it can hold is $2^{31} - 1$, which is about 2 billion. A `long` uses 64 bits, so the largest value is $2^{63} - 1$, which is about 9 quintillion.

Take a minute to read the documentation for these and other methods in the `Math` class. The easiest way to find documentation for Java classes is to do a web search for “Java” and the name of the class.

Composition

You have probably learned how to evaluate simple expressions like $\sin(\pi/2)$ and $\log(1/x)$. First, you evaluate the expression in parentheses, which is the argument of the function. Then you can evaluate the function itself, either by hand or by punching it into a calculator.

This process can be applied repeatedly to evaluate more-complex expressions like $\log(1/\sin(\pi/2))$. First we evaluate the argument of the innermost function ($\pi/2 = 1.57$), then evaluate the function itself ($\sin(1.57) = 1.0$), and so on.

Just as with mathematical functions, Java methods can be **composed** to solve complex problems. That means you can use one method as part of another. In fact, you can use any expression as an argument to a method, as long as the resulting value has the correct type:

```
double x = Math.cos(angle + Math.PI / 2.0);
```

This statement divides `Math.PI` by 2, adds the result to `angle`, and computes the cosine of the sum. You can also take the result of one method and pass it as an argument to another:

```
double x = Math.exp(Math.log(10.0));
```

In Java, the `log` method always uses base e . So this statement finds the log base e of 10, and then raises e to that power. The result gets assigned to `x`.

Some math methods take more than one argument. For example, `Math.pow` takes two arguments and raises the first to the power of the second. This line computes 2^{10} and assigns the value `1024.0` to the variable `x`:

```
double x = Math.pow(2.0, 10.0);
```

When using `Math` methods, beginners often forget the word `Math`. For example, if you just write `x = pow(2.0, 10.0)`, you will get a compiler error:

```
File: Test.java [line: 5]
Error: cannot find symbol
  symbol:   method pow(double,double)
  location: class Test
```

The message `cannot find symbol` is confusing, but the last two lines provide a useful hint. The compiler is looking for a method named `pow` in the file `Test.java` (the file for this example). If you don't specify a class name when referring to a method, the compiler looks in the current class by default.

Return Values

When you invoke a `void` method, the invocation is usually on a line all by itself. For example:

```
printTime(hour + 1, 0);
```

On the other hand, when you invoke a value-returning method, you have to do something with the return value. We usually assign it to a variable or use it as part of an expression, like this:

```
double error = Math.abs(expect - actual);  
double height = radius * Math.sin(angle);
```

Compared to `void` methods, value-returning methods differ in two ways:

- They declare the type of the return value (the **return type**).
- They use at least one return statement to provide a **return value**.

Here's an example from a program named `Circle.java`. The `calculateArea` method takes a `double` as a parameter and returns the area of a circle with that radius (i.e., πr^2):

```
public static double calculateArea(double radius) {  
    double result = Math.PI * radius * radius;  
    return result;  
}
```

As usual, this method is `public` and `static`. But in the place where we are used to seeing `void`, we see `double`, which means that the return value from this method is a `double`.

The last line is a new form of the `return` statement that means, “Return immediately from this method, and use the following expression as the return value.” The expression you provide can be arbitrarily complex, so we could have written this method more concisely:

```
public static double calculateArea(double radius) {  
    return Math.PI * radius * radius;  
}
```

On the other hand, **temporary variables** like `result` often make debugging easier, especially when you are stepping through code by using an interactive debugger (see “Tracing with a Debugger” on page 265).

Figure 4-2 illustrates how data values flows through the program. When the `main` method invokes `calculateArea`, the value `5.0` is assigned to the parameter `radius`. `calculateArea` then returns the value `78.54`, which is assigned to the variable `area`.

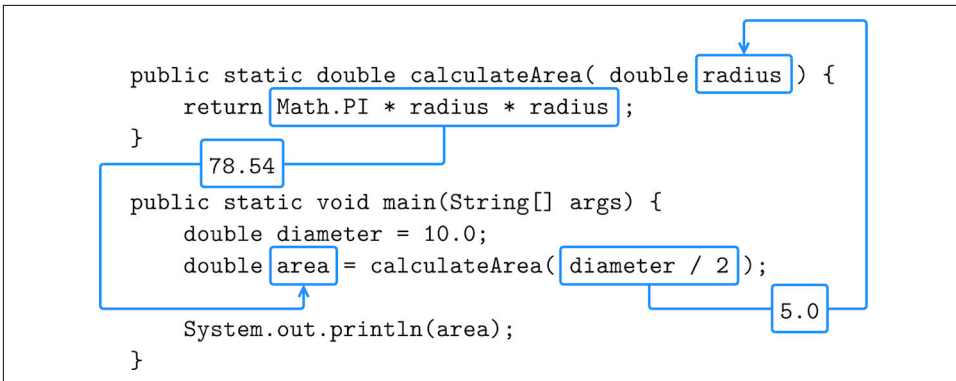


Figure 4-2. Passing a parameter and saving the return value

The type of the expression in the return statement must match the return type of the method itself. When you declare that the return type is double, you are making a promise that this method will eventually produce a double value. If you try to return with no expression, or return an expression with the wrong type, the compiler will give an error.

Incremental Development

People often make the mistake of writing a lot of code before they try to compile and run it. Then they spend way too much time debugging. A better approach is what we call **incremental development**. Its key aspects are as follows:

- Start with a working program and make small, incremental changes. At any point, if there is an error, you will know where to look.
- Use variables to hold intermediate values so you can check them, either with print statements or by using a debugger.
- Once the program is working, you can consolidate multiple statements into compound expressions (but only if it does not make the program more difficult to read).

As an example, suppose you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2) . By the usual definition:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a distance method should look like in Java. In other words, what are the inputs (parameters) and what is the output (return value)? For

this method, the parameters are the two points, and it is natural to represent them using four double values. The return value is the distance, which should also have type double.

Already we can write an outline for the method, which is sometimes called a **stub**. The stub includes the method declaration and a return statement:

```
public static double distance
    (double x1, double y1, double x2, double y2) {
    return 0.0; // stub
}
```

The return statement is a placeholder that is necessary only for the program to compile. At this stage, the program doesn't do anything useful, but it is good to compile it so we can find any syntax errors before we add more code.

It's usually a good idea to think about testing *before* you develop new methods; doing so can help you figure out how to implement them. To test the method, we can invoke it from `main` by using the sample values:

```
double dist = distance(1.0, 2.0, 4.0, 6.0);
```

With these values, the horizontal distance is 3.0 and the vertical distance is 4.0. So the result should be 5.0, the hypotenuse of a 3-4-5 triangle. When you are testing a method, it is necessary to know the right answer.

Once we have compiled the stub, we can start adding code one line at a time. After each incremental change, we recompile and run the program. If there is an error, we have a good idea of where to look: the lines we just added.

The next step is to find the differences: $x_2 - x_1$ and $y_2 - y_1$. We store those values in temporary variables named `dx` and `dy`, so that we can examine them with print statements before proceeding. They should be 3.0 and 4.0:

```
public static double distance
    (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    System.out.println("dx is " + dx);
    System.out.println("dy is " + dy);
    return 0.0; // stub
}
```

We will remove the print statements when the method is finished. Code like that is called **scaffolding**, because it is helpful for building the program but is not part of the final product.

The next step is to square `dx` and `dy`. We could use the `Math.pow` method, but it is simpler (and more efficient) to multiply each term by itself.

Then we add the squares and print the result so far:

```
public static double distance
    (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx * dx + dy * dy;
    System.out.println("dsquared is " + dsquared);
    return 0.0; // stub
}
```

Again, you should compile and run the program at this stage and check the intermediate value, which should be 25.0. Finally, we can use `Math.sqrt` to compute and return the result:

```
public static double distance
    (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx * dx + dy * dy;
    double result = Math.sqrt(dsquared);
    return result;
}
```

As you gain more experience programming, you might write and debug more than one line at a time. But if you find yourself spending a lot of time debugging, consider taking smaller steps.

Vocabulary

void

A special return type indicating the method does not return a value.

invoke

To cause a method to execute. Also known as *calling* a method.

flow of execution

The order in which Java executes methods and statements. It may not necessarily be from top to bottom in the source file.

argument

A value that you provide when you call a method. This value must have the type that the method expects.

parameter

A piece of information that a method requires before it can run. Parameters are variables: they contain values and have types.

parameter passing

The process of assigning an argument value to a parameter variable.

local variable

A variable declared inside a method. Local variables cannot be accessed from outside their method.

stack diagram

A graphical representation of the variables belonging to each method. The method calls are *stacked* from top to bottom, in the flow of execution.

frame

In a stack diagram, a representation of the variables and parameters for a method, along with their current values.

scope

The area of a program where a variable can be used.

compose

To combine simple expressions and statements into compound expressions and statements.

return type

The type of value a method returns.

return value

The value provided as the result of a method invocation.

temporary variable

A short-lived variable, often used for debugging.

incremental development

A process for creating programs by writing a few lines at a time, compiling, and testing.

stub

A placeholder for an incomplete method so that the class will compile.

scaffolding

Code that is used during program development but is not part of the final version.

Exercises

The code for this chapter is in the *ch04* directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

If you have not already read “[Command-Line Testing](#)” on page 262, now might be a good time. It describes an efficient way to test programs that take input from the user and display specific output.

Exercise 4-1.

The purpose of this exercise is to take code from a previous exercise and redesign it as a method that takes parameters. Start with a working solution to [Exercise 2-2](#).

1. Write a method called `printAmerican` that takes the day, date, month, and year as parameters and displays them in American format.
2. Test your method by invoking it from `main` and passing appropriate arguments. The output should look something like this (except the date might be different):
Saturday, July 22, 2015
3. Once you have debugged `printAmerican`, write another method called `printEuropean` that displays the date in European format.

Exercise 4-2.

This exercise reviews the flow of execution through a program with multiple methods. Read the following code and answer the questions:

```
public static void main(String[] args) {
    zippo("rattle", 13);
}

public static void baffle(String blimp) {
    System.out.println(blimp);
    zippo("ping", -5);
}

public static void zippo(String quince, int flag) {
    if (flag < 0) {
        System.out.println(quince + " zoop");
    } else {
        System.out.println("ik");
        baffle(quince);
        System.out.println("boo-wa-ha-ha");
    }
}
```

1. Write the number 1 next to the first line of code in this program that will execute.
2. Write the number 2 next to the second line of code, and so on until the end of the program. If a line is executed more than once, it might end up with more than one number next to it.

3. What is the value of the parameter `blimp` when `baffle` gets invoked?
4. What is the output of this program?

Exercise 4-3.

Answer the following questions without running the program on a computer.

1. Draw a stack diagram that shows the state of the program the first time `ping` is invoked.
2. What is output by the following program? Be precise about the placement of spaces and newlines.

```
public static void zoop() {
    baffle();
    System.out.print("You wugga ");
    baffle();
}

public static void main(String[] args) {
    System.out.print("No, I ");
    zoop();
    System.out.print("I ");
    baffle();
}

public static void baffle() {
    System.out.print("wug");
    ping();
}

public static void ping() {
    System.out.println(".");
}
```

Exercise 4-4.

If you have a question about whether something is legal, and what happens if it is not, a good way to find out is to ask the compiler. Answer the following questions by trying them out.

1. What happens if you invoke a value method and don't do anything with the result; that is, if you don't assign it to a variable or use it as part of a larger expression?
2. What happens if you use a void method as part of an expression? For example, try `System.out.println("boo!") + 7;`.

Exercise 4-5.

Draw a stack diagram that shows the state of the program the *second* time `zoop` is invoked. What is the complete output?

```
public static void zoop(String fred, int bob) {
    System.out.println(fred);
    if (bob == 5) {
        ping("not ");
    } else {
        System.out.println("!");
    }
}

public static void main(String[] args) {
    int bizz = 5;
    int buzz = 2;
    zoop("just for", bizz);
    clink(2 * buzz);
}

public static void clink(int fork) {
    System.out.print("It's ");
    zoop("breakfast ", fork);
}

public static void ping(String strangStrung) {
    System.out.println("any " + strangStrung + "more ");
}
```

Exercise 4-6.

Many computations can be expressed more concisely using the *multadd* operation, which takes three operands and computes $a * b + c$. Some processors even provide a hardware implementation of this operation for floating-point numbers.

1. Create a new program called `Multadd.java`.
2. Write a method called `multadd` that takes three doubles as parameters and returns $a * b + c$.
3. Write a `main` method that tests `multadd` by invoking it with a few simple parameters, like `1.0`, `2.0`, `3.0`.
4. Also in `main`, use `multadd` to compute the following values:

$$\sin \frac{\pi}{4} + \frac{\cos \frac{\pi}{4}}{2}$$

$$\log 10 + \log 20$$

5. Write a method called `expSum` that takes a `double` as a parameter and uses `multadd` to calculate:

$$xe^{-x} + \sqrt{1 - e^{-x}}$$

Hint: The method for raising e to a power is `Math.exp`.

In the last part of this exercise, you need to write a method that invokes another method you wrote. Whenever you do that, it is a good idea to test the first method carefully before working on the second. Otherwise, you might find yourself debugging two methods at the same time, which can be difficult.

One of the purposes of this exercise is to practice pattern-matching: the ability to recognize a specific problem as an instance of a general category of problems.

Conditionals and Logic

The programs in the previous chapters do pretty much the same thing every time, regardless of the input. For more-complex computations, programs usually react to inputs, check for certain conditions, and generate applicable results. This chapter introduces Java language features for expressing logic and making decisions.

Relational Operators

Java has six **relational operators** that test the relationship between two values (e.g., whether they are equal, or whether one is greater than the other). The following expressions show how they are used:

```
x == y      // x is equal to y
x != y      // x is not equal to y
x > y       // x is greater than y
x < y       // x is less than y
x >= y      // x is greater than or equal to y
x <= y      // x is less than or equal to y
```

The result of a relational operator is one of two special values: `true` or `false`. These values belong to the data type **boolean**, named after the mathematician George Boole. He developed an algebraic way of representing logic.

You are probably familiar with these operators, but notice how Java is different from mathematical symbols like $=$, \neq , and \geq . A common error is to use a single `=` instead of a double `==` when comparing values. Remember that `=` is the *assignment* operator, and `==` is a *relational* operator. Also, the operators `=<` and `=>` do not exist.

The two sides of a relational operator have to be compatible. For example, the expression `5 < "6"` is invalid because `5` is an `int` and `"6"` is a `String`. When comparing values of different numeric types, Java applies the same conversion rules you saw

previously with the assignment operator. For example, when evaluating the expression `5 < 6.0`, Java automatically converts the 5 to `5.0`.

The if-else Statement

To write useful programs, we almost always need to check conditions and react accordingly. **Conditional statements** give us this ability. The simplest conditional statement in Java is the `if` statement:

```
if (x > 0) {
    System.out.println("x is positive");
}
```

The expression in parentheses is called the *condition*. If it is true, the statements in braces get executed. If the condition is false, execution skips over that **block** of code. The condition in parentheses can be any boolean expression.

A second form of conditional statement has two possibilities, indicated by `if` and `else`. The possibilities are called **branches**, and the condition determines which branch gets executed:

```
if (x % 2 == 0) {
    System.out.println("x is even");
} else {
    System.out.println("x is odd");
}
```

If the remainder when `x` is divided by 2 is 0, we know that `x` is even, and the program displays a message to that effect. If the condition is false, the second print statement is executed instead. Since the condition must be true or false, exactly one of the branches will run.

The braces are optional for branches that have only one statement. So we could have written the previous example this way:

```
if (x % 2 == 0)
    System.out.println("x is even");
else
    System.out.println("x is odd");
```

However, it's better to use braces—even when they are optional—to avoid making the mistake of adding statements to a one-line `if` or `else` block. This code is misleading because it's not indented correctly:

```
if (x > 0)
    System.out.println("x is positive");
    System.out.println("x is not zero");
```

Since there are no braces, only the first `println` is part of the `if` statement. Here is what the compiler actually sees:

```
if (x > 0) {
    System.out.println("x is positive");
}
    System.out.println("x is not zero");
```

As a result, the second `println` runs no matter what. Even experienced programmers make this mistake; search the web for Apple’s “goto fail” bug.

In all previous examples, notice that there is no semicolon at the end of the `if` or `else` lines. Instead, a new block should be defined using braces. Another common mistake is to put a semicolon after the condition, like this:

```
int x = 1;
if (x % 2 == 0); { // incorrect semicolon
    System.out.println("x is even");
}
```

This code will compile, but the program will output “x is even” regardless of the value of `x`. Here is the same incorrect code with better formatting:

```
int x = 1;
if (x % 2 == 0)
    ; // empty statement
{
    System.out.println("x is even");
}
```

Because of the semicolon, the `if` statement compiles as if there are no braces, and the subsequent block runs independently. As a general rule, each line of Java code should end with a semicolon or brace—but not both.

The compiler won’t complain if you omit optional braces or write empty statements. Doing so is allowed by the Java language, but it often results in bugs that are difficult to find. Development tools like Checkstyle (see “[Running Checkstyle](#)” on page 264) can warn you about these and other kinds of programming mistakes.

Chaining and Nesting

Sometimes you want to check related conditions and choose one of several actions. One way to do this is by **chaining** a series of `if` and `else` blocks:

```
if (x > 0) {
    System.out.println("x is positive");
} else if (x < 0) {
    System.out.println("x is negative");
} else {
    System.out.println("x is zero");
}
```

These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements and braces lined up, you are less likely to make syntax errors.

Notice that the last branch is simply `else`, not `else if (x == 0)`. At this point in the chain, we know that `x` is not positive and `x` is not negative. There is no need to test whether `x` is 0, because there is no other possibility.

In addition to chaining, you can also make complex decisions by **nesting** one conditional statement inside another. We could have written the previous example as follows:

```
if (x > 0) {
    System.out.println("x is positive");
} else {
    if (x < 0) {
        System.out.println("x is negative");
    } else {
        System.out.println("x is zero");
    }
}
```

The outer conditional has two branches. The first branch contains a print statement, and the second branch contains another conditional statement, which has two branches of its own. These two branches are also print statements, but they could have been conditional statements as well.

These kinds of nested structures are common, but they can become difficult to read very quickly. Good indentation is essential to make the structure (or intended structure) apparent to the reader.

The switch Statement

If you need to make a series of decisions, chaining `else if` blocks can get long and redundant. For example, consider a program that converts integers like 1, 2, and 3 into words like "one", "two", and "three":

```
if (number == 1) {
    word = "one";
} else if (number == 2) {
    word = "two";
} else if (number == 3) {
    word = "three";
} else {
    word = "unknown";
}
```

This chain could go on and on, especially for banking programs that write numbers in long form (e.g., “one hundred twenty-three and 45/100 dollars”). An alternative way to evaluate many possible values of an expression is to use a `switch` statement:

```
switch (number) {
    case 1:
        word = "one";
        break;
    case 2:
        word = "two";
        break;
    case 3:
        word = "three";
        break;
    default:
        word = "unknown";
        break;
}
```

The body of a `switch` statement is organized into one or more `case` blocks. Each `case` ends with a `break` statement, which exits the `switch` body. The `default` block is optional and executed only if none of the cases apply.

Although `switch` statements appear longer than chained `else if` blocks, they are particularly useful when multiple cases can be grouped:

```
switch (food) {
    case "apple":
    case "banana":
    case "cherry":
        System.out.println("Fruit!");
        break;
    case "asparagus":
    case "broccoli":
    case "carrot":
        System.out.println("Vegetable!");
        break;
}
```

Logical Operators

In addition to the relational operators, Java also has three **logical operators**: `&&`, `||`, and `!`, which respectively stand for *and*, *or*, and *not*. The results of these operators are similar to their meanings in English.

For example, `x > 0 && x < 10` is true when `x` is both greater than 0 *and* less than 10.

The expression `x < 0 || x > 10` is true if either condition is true; that is, if `x` is less than 0 *or* greater than 10.

Finally, `!(x > 0)` is true if `x` is *not* greater than 0. The parentheses are necessary in this example because in the order of operations `!` comes before `>`.

In order for an expression with `&&` to be true, both sides of the `&&` operator must be true. And in order for an expression with `||` to be false, both sides of the `||` operator must be false.

The `&&` operator can be used to simplify nested `if` statements. For example, the following code can be rewritten with a single condition:

```
if (x == 0) {
    if (y == 0) {
        System.out.println("Both x and y are zero");
    }
}

// combined
if (x == 0 && y == 0) {
    System.out.println("Both x and y are zero");
}
```

Likewise, the `||` operator can simplify chained `if` statements. Since the branches are the same, there is no need to duplicate that code:

```
if (x == 0) {
    System.out.println("Either x or y is zero");
} else if (y == 0) {
    System.out.println("Either x or y is zero");
}

// combined
if (x == 0 || y == 0) {
    System.out.println("Either x or y is zero");
}
```

Then again, if the statements in the branches were different, we could not combine them into one block. But it's useful to explore different ways of representing the same logic, especially when it's complex.

Logical operators evaluate the second expression only when necessary. For example, `true || anything` is always true, so Java does not need to evaluate the expression `anything`. Likewise, `false && anything` is always false.

Ignoring the second operand, when possible, is called **short-circuit evaluation**, by analogy with an electrical circuit. Short-circuit evaluation can save time, especially if `anything` takes a long time to evaluate. It can also avoid unnecessary errors, if `anything` might fail.

De Morgan's Laws

Sometimes you need to negate an expression containing a mix of relational and logical operators. For example, to test if x and y are both nonzero, you could write the following:

```
if (!(x == 0 || y == 0)) {  
    System.out.println("Neither x nor y is zero");  
}
```

This condition is difficult to read because of the $!$ and parentheses. A better way to negate logic expressions is to apply **De Morgan's laws**:

- $!(A \ \&\& \ B)$ is the same as $!A \ || \ !B$
- $!(A \ || \ B)$ is the same as $!A \ \&\& \ !B$

In words, negating a logical expression is the same as negating each term and changing the operator. The $!$ operator takes precedence over $\&\&$ and $||$, so you don't have to put parentheses around the individual terms $!A$ and $!B$.

De Morgan's laws also apply to the relational operators. In this case, negating each term means using the "opposite" relational operator:

- $!(x < 5 \ \&\& \ y == 3)$ is the same as $x >= 5 \ || \ y != 3$
- $!(x >= 1 \ || \ y != 7)$ is the same as $x < 1 \ \&\& \ y == 7$

It may help to read these examples out loud in English. For instance, "If I don't want the case where x is less than 5 and y is 3, then I need x to be greater than or equal to 5, or I need y to be anything but 3."

Returning to the previous example, here is the revised condition. In English, it reads, "If x is not zero and y is not zero." The logic is the same, and the source code is easier to read:

```
if (x != 0 && y != 0) {  
    System.out.println("Neither x nor y is zero");  
}
```

Boolean Variables

To store a true or false value, you need a boolean variable. You can declare and assign them like other variables. In this example, the first line is a variable declaration, the second is an assignment, and the third is both:

```
boolean flag;  
flag = true;  
boolean testResult = false;
```

Since relational and logical operators evaluate to a boolean value, you can store the result of a comparison in a variable:

```
boolean evenFlag = (n % 2 == 0);    // true if n is even
boolean positiveFlag = (x > 0);    // true if x is positive
```

The parentheses are unnecessary, but they make the code easier to understand. A variable defined in this way is called a **flag**, because it signals, or *flags*, the presence or absence of a condition.

You can use flag variables as part of a conditional statement:

```
if (evenFlag) {
    System.out.println("n was even when I checked it");
}
```

Flags may not seem that useful at this point, but they will help simplify complex conditions later. Each part of a condition can be stored in a separate flag, and these flags can be combined with logical operators.

Notice that we didn't have to write `if (evenFlag == true)`. Since `evenFlag` is a boolean, it's already a condition. To check if a flag is false, we simply negate the flag:

```
if (!evenFlag) {
    System.out.println("n was odd when I checked it");
}
```

In general, you should never compare anything to `true` or `false`. Doing so makes the code more verbose and awkward to read out loud.

Boolean Methods

Methods can return boolean values, just like any other type, which is often convenient for hiding tests inside methods. For example:

```
public static boolean isSingleDigit(int x) {
    if (x > -10 && x < 10) {
        return true;
    } else {
        return false;
    }
}
```

The name of this method is `isSingleDigit`. It is common to give boolean methods names that sound like yes/no questions. Since the return type is `boolean`, the return statement has to provide a boolean expression.

The code itself is straightforward, although it is longer than it needs to be. Remember that the expression `x > -10 && x < 10` has type `boolean`, so there is nothing wrong with returning it directly (without the `if` statement):


```
public static boolean isSingleDigit(int x) {  
    return x > -10 && x < 10;  
}
```

In main, you can invoke the method in the usual ways:

```
System.out.println(isSingleDigit(2));  
boolean bigFlag = !isSingleDigit(17);
```

The first line displays true because 2 is a single-digit number. The second line sets bigFlag to true, because 17 is *not* a single-digit number.

Conditional statements often invoke boolean methods and use the result as the condition:

```
if (isSingleDigit(z)) {  
    System.out.println("z is small");  
} else {  
    System.out.println("z is big");  
}
```

Examples like this one almost read like English: “If is single digit z, print z is small else print z is big.”

Validating Input

One of the most important tasks in any computer program is to **validate** input from the user. People often make mistakes while typing, especially on smartphones, and incorrect inputs may cause your program to fail.

Even worse, someone (i.e., a **hacker**) may intentionally try to break into your system by entering unexpected inputs. You should never assume that users will input the right kind of data.

Consider this simple program that prompts the user for a number and computes its logarithm:

```
Scanner in = new Scanner(System.in);  
System.out.print("Enter a number: ");  
double x = in.nextDouble();  
double y = Math.log(x);  
System.out.println("The log is " + y);
```

In mathematics, the natural logarithm (base e) is undefined when $x \leq 0$. In Java, if you ask for `Math.log(-1)`, it returns **NaN**, which stands for *not a number*. We can check for this condition and print an appropriate message:

```

if (x > 0) {
    double y = Math.log(x);
    System.out.println("The log is " + y);
} else {
    System.out.println("The log is undefined");
}

```

The output is better now, but there is another problem. What if the user doesn't enter a number at all? What would happen if they typed the word `hello`, either by accident or on purpose?

```

Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextDouble(Scanner.java:2413)
    at Logarithm.main(Logarithm.java:8)

```

If the user inputs a `String` when we expect a `double`, Java reports an `input mismatch` exception. We can prevent this run-time error from happening by testing the input first.

The `Scanner` class provides `hasNextDouble`, which checks whether the next input can be interpreted as a `double`. If not, we can display an error message:

```

if (!in.hasNextDouble()) {
    String word = in.next();
    System.err.println(word + "is not a number");
}

```

In contrast to `in.nextLine`, which returns an entire line of input, the `in.next` method returns only the next token of input. We can use `in.next` to show the user exactly which word they typed was not a number.

This example also uses `System.err`, which is an `OutputStream` for error messages and warnings. Some development environments display output to `System.err` with a different color or in a separate window.

Example Program

In this chapter, you have seen relational and logical operators, `if` statements, boolean methods, and validating input. The following program shows how the individual code examples in the previous section fit together:

```

import java.util.Scanner;

/**
 * Demonstrates input validation using if statements.
 */
public class Logarithm {

```

```

public static void main(String[] args) {

    // prompt for input
    Scanner in = new Scanner(System.in);
    System.out.print("Enter a number: ");

    // check the format
    if (!in.hasNextDouble()) {
        String word = in.next();
        System.err.println(word + " is not a number");
        return;
    }

    // check the range
    double x = in.nextDouble();
    if (x > 0) {
        double y = Math.log(x);
        System.out.println("The log is " + y);
    } else {
        System.out.println("The log is undefined");
    }
}
}

```

Notice that the `return` statement allows you to exit a method before you reach the end of it. Returning from `main` terminates the program.

What started as five lines of code at the beginning of “Validating Input” on page 71 is now a 30-line program. Making programs robust (and secure) often requires a lot of additional checking, as shown in this example.

It’s important to write comments every few lines to make your code easier to understand. Comments not only help other people read your code, but also help you document what you’re trying to do. If there’s a mistake in the code, finding it will be a lot easier when there are good comments.

Vocabulary

relational operator

An operator that compares two values and produces a `boolean` indicating the relationship between them.

boolean

A data type with only two possible values, `true` and `false`.

conditional statement

A statement that uses a condition to determine which statements to execute.

block

A sequence of statements, surrounded by braces, that generally runs as the result of a condition.

branch

One of the alternative blocks after a conditional statement. For example, an `if-else` statement has two branches.

chaining

A way of joining several conditional statements in sequence.

nesting

Putting a conditional statement inside one or both branches of another conditional statement.

logical operator

An operator that combines boolean values and produces a boolean value.

short-circuit evaluation

A way of evaluating logical operators that evaluates the second operand only if necessary.

De Morgan's laws

Mathematical rules that show how to negate a logical expression.

flag

A variable (usually `boolean`) that represents a condition or status.

validate

To confirm that an input value is of the correct type and within the expected range.

hacker

A programmer who breaks into computer systems. The term *hacker* may also apply to someone who enjoys writing code.

NaN

A special floating-point value that stands for *not a number*.

Exercises

The code for this chapter is in the `ch05` directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

If you have not already read “Running Checkstyle” on page 264, now might be a good time. It describes Checkstyle, a tool that analyzes many aspects of your source code.

Exercise 5-1.

Rewrite the following code by using a single `if` statement:

```
if (x > 0) {  
    if (x < 10) {  
        System.out.println("positive single digit number.");  
    }  
}
```

Exercise 5-2.

Now that we have conditional statements, we can get back to the Guess My Number game from [Exercise 3-4](#).

You should already have a program that chooses a random number, prompts the user to guess it, and displays the difference between the guess and the chosen number.

By adding a small amount of code at a time and testing as you go, modify the program so it tells the user whether the guess is too high or too low, and then prompts the user for another guess.

The program should continue until the user gets it right or guesses incorrectly three times. If the user guesses the correct number, display a message and terminate the program.

Exercise 5-3.

Fermat's Last Theorem says that there are no integers a , b , c , and n such that $a^n + b^n = c^n$, except when $n \leq 2$.

Write a program named `Fermat.java` that inputs four integers (a , b , c , and n) and checks to see if Fermat's theorem holds. If n is greater than 2 and $a^n + b^n = c^n$, the program should display `Holy smokes, Fermat was wrong!` Otherwise, the program should display `"No, that doesn't work."`

Hint: You might want to use `Math.pow`.

Exercise 5-4.

Using the following variables, evaluate the logic expressions in the table that follows. Write your answers as true, false, or error.

```
boolean yes = true;
boolean no = false;
int loVal = -999;
int hiVal = 999;
double grade = 87.5;
double amount = 50.0;
String hello = "world";
```

Expression	Result
yes == no grade > amount	
amount == 40.0 50.0	
hiVal != loVal loVal < 0	
True hello.length() > 0	
hello.isEmpty() && yes	
grade <= 100 && !false	
!yes no	
grade > 75 > amount	
amount <= hiVal && amount >= loVal	
no && !no yes && !yes	

Exercise 5-5.

What is the output of the following program? Determine the answer without using a computer. The purpose of this exercise is to make sure you understand logical operators and the flow of execution through methods.

```
public static void main(String[] args) {
    boolean flag1 = isHoopy(202);
    boolean flag2 = isFrabjuous(202);
    System.out.println(flag1);
    System.out.println(flag2);
    if (flag1 && flag2) {
        System.out.println("ping!");
    }
    if (flag1 || flag2) {
        System.out.println("pong!");
    }
}
```

```

public static boolean isHoopy(int x) {
    boolean hoopyFlag;
    if (x % 2 == 0) {
        hoopyFlag = true;
    } else {
        hoopyFlag = false;
    }
    return hoopyFlag;
}

public static boolean isFrabjuous(int x) {
    boolean frabjuousFlag;
    if (x > 0) {
        frabjuousFlag = true;
    } else {
        frabjuousFlag = false;
    }
    return frabjuousFlag;
}

```

Exercise 5-6.

Write a program named `Quadratic.java` that finds the roots of $ax^2 + bx + c = 0$ using the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Prompt the user to input integers for a , b , and c . Compute the two solutions for x , and display the results.

Your program should be able to handle inputs for which there is only one or no solution. Specifically, it should not divide by zero or take the square root of a negative number.

Be sure to validate all inputs. The user should never see an input mismatch exception. Display specific error messages that include the invalid input.

Exercise 5-7.

If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are 1 inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:

If any of the three lengths is greater than the sum of the other two, you cannot form a triangle.

Write a program named `Triangle.java` that inputs three integers, and then outputs whether you can (or cannot) form a triangle from the given lengths. Reuse your code from the previous exercise to validate the inputs. Display an error if any of the lengths are negative or zero.

Loops and Strings

Computers are often used to automate repetitive tasks, such as searching for text in documents. Repeating tasks without making errors is something that computers do well and people do poorly.

In this chapter, you'll learn how to use `while` and `for` loops to add repetition to your code. We'll also take a first look at `String` methods and solve some interesting problems.

The while Statement

Using a `while` statement, we can repeat the same code multiple times:

```
int n = 3;
while (n > 0) {
    System.out.println(n);
    n = n - 1;
}
System.out.println("Blastoff!");
```

Reading the code in English sounds like this: “Start with `n` set to 3. While `n` is greater than 0, print the value of `n`, and reduce the value of `n` by 1. When you get to 0, print `Blastoff!` So the output is as follows:

```
3
2
1
Blastoff!
```

The flow of execution for a `while` statement is shown here:

1. Evaluate the condition in parentheses, yielding `true` or `false`.
2. If the condition is `false`, skip the following statements in braces.
3. If the condition is `true`, execute the statements and go back to step 1.

This type of flow is called a **loop**, because the last step “loops back around” to the first. [Figure 6-1](#) shows this idea using a flowchart.

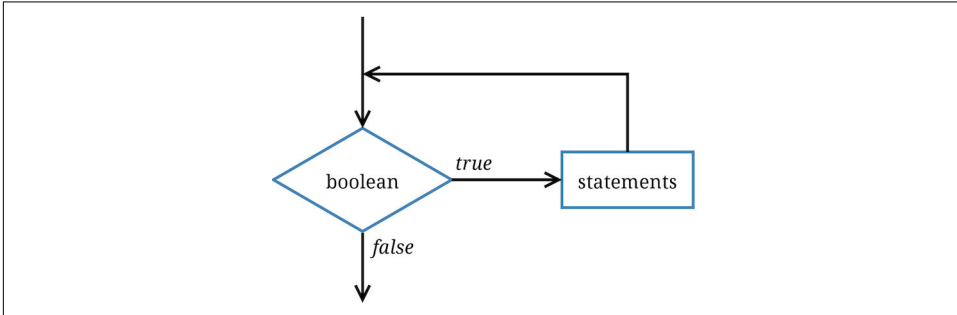


Figure 6-1. Flow of execution for a while loop

The **body** of the loop should change the value of one or more variables so that, eventually, the condition becomes `false` and the loop terminates. Otherwise, the loop will repeat forever, which is called an **infinite loop**:

```
int n = 3;
while (n > 0) {
    System.out.println(n);
    // n never changes
}
```

This example will print the number 3 forever, or at least until you terminate the program. An endless source of amusement for computer scientists is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an infinite loop.

In the first example, we can prove that the loop terminates when `n` is positive. But in general, it is not so easy to tell whether a loop terminates. For example, this loop continues until `n` is 1 (which makes the condition `false`):

```
while (n != 1) {
    System.out.println(n);
    if (n % 2 == 0) { // n is even
        n = n / 2;
    } else { // n is odd
        n = 3 * n + 1;
    }
}
```

Each time through the loop, the program displays the value of n and then checks whether it is even or odd. If it is even, the value of n is divided by 2. If it is odd, the value is replaced by $3n + 1$. For example, if the starting value is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since n sometimes increases and sometimes decreases, there is no obvious proof that n will ever reach 1 and that the program will ever terminate. For some values of n , such as the powers of two, we can prove that it terminates. The previous example ends with such a sequence, starting when n is 16 (or 2^4).

The hard question is whether this program terminates for *all* values of n . So far, no one has been able to prove it *or* disprove it! For more information, see [the “Collatz conjecture” entry on Wikipedia](#).

Increment and Decrement

Here is another `while` loop example; this one displays the numbers 1 to 5:

```
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i++; // add 1 to i
}
```

Assignments like `i = i + 1` don't often appear in loops, because Java provides a more concise way to add and subtract by one. Specifically, `++` is the **increment** operator; it has the same effect as `i = i + 1`. And `--` is the **decrement** operator; it has the same effect as `i = i - 1`.

If you want to increment or decrement a variable by an amount other than 1, you can use `+=` and `-=`. For example, `i += 2` increments `i` by 2:

```
int i = 2;
while (i <= 8) {
    System.out.print(i + ", ");
    i += 2; // add 2 to i
}
System.out.println("Who do we appreciate?");
```

And the output is as follows:

```
2, 4, 6, 8, Who do we appreciate?
```

The for Statement

The loops we have written so far have three parts in common. They start by initializing a variable, they have a condition that depends on that variable, and they do something inside the loop to update that variable.

Running the same code multiple times is called **iteration**. It's so common that there is another statement, the `for` loop, that expresses it more concisely. For example, we can rewrite the 2-4-6-8 loop this way:

```
for (int i = 2; i <= 8; i += 2) {  
    System.out.print(i + ", ");  
}  
System.out.println("Who do we appreciate?");
```

`for` loops have three components in parentheses, separated by semicolons: the initializer, the condition, and the update:

1. The *initializer* runs once at the very beginning of the loop. It is equivalent to the line before the `while` statement.
2. The *condition* is checked each time through the loop. If it is `false`, the loop ends. Otherwise, the body of the loop is executed (again).
3. At the end of each iteration, the *update* runs, and we go back to step 2.

The `for` loop is often easier to read because it puts all the loop-related statements at the top of the loop. Doing so allows you to focus on the statements inside the loop body. [Figure 6-2](#) illustrates `for` loops with a flowchart.

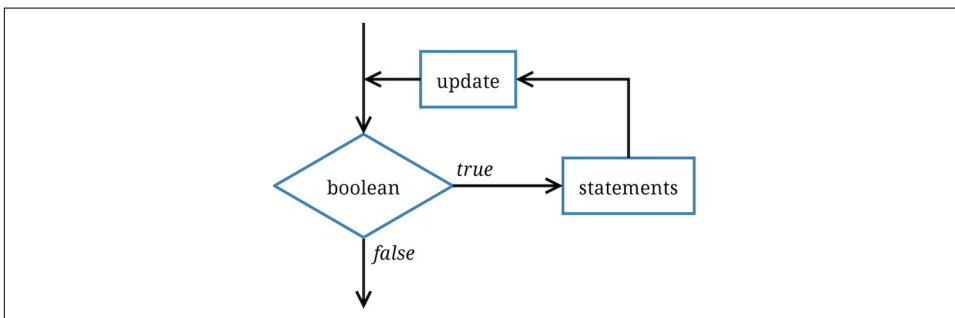


Figure 6-2. Flow of execution for a `for` loop

There is another difference between `for` loops and `while` loops: if you declare a variable in the initializer, it exists only *inside* the `for` loop. For example:

```
for (int n = 3; n > 0; n--) {  
    System.out.println(n);  
}  
System.out.println("n is now " + n); // compiler error
```

The last line tries to display `n` (for no reason other than demonstration), but it won't work. If you need to use a loop variable outside the loop, you have to declare it *outside* the loop, like this:

```

int n;
for (n = 3; n > 0; n--) {
    System.out.println(n);
}
System.out.println("n is now " + n);

```

Notice that the for statement does not say `int n = 3`. Rather, it simply initializes the existing variable `n`.

Nested Loops

Like conditional statements, loops can be nested one inside the other. Nested loops allow you to iterate over two variables. For example, we can generate a “multiplication table” like this:

```

for (int x = 1; x <= 10; x++) {
    for (int y = 1; y <= 10; y++) {
        System.out.printf("%4d", x * y);
    }
    System.out.println();
}

```

Variables like `x` and `y` are called **loop variables**, because they control the execution of a loop. In this example, the first loop (for `x`) is known as the *outer loop*, and the second loop (for `y`) is known as the *inner loop*.

Each loop repeats its corresponding statements 10 times. The outer loop iterates from 1 to 10 only once, but the inner loop iterates from 1 to 10 each of those 10 times. As a result, the `printf` method is invoked 100 times.

The format specifier `%4d` displays the value of `x * y` padded with spaces so it’s four characters wide. Doing so causes the output to align vertically, regardless of how many digits the numbers have:

```

1  2  3  4  5  6  7  8  9 10
2  4  6  8 10 12 14 16 18 20
3  6  9 12 15 18 21 24 27 30
4  8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100

```

It’s important to realize that the output is displayed row by row. The inner loop displays a single row of output, followed by a newline. The outer loop iterates over the rows themselves. Another way to read nested loops, like the ones in this example, is, “For each row `x`, and for each column `y`, ...”

Characters

Some of the most interesting problems in computer science involve searching and manipulating text. In the next few sections, we'll discuss how to apply loops to strings. Although the examples are short, the techniques work the same whether you have one word or one million words.

Strings provide a method named `charAt`. It returns a `char`, a data type that stores an individual character (as opposed to strings of them):

```
String fruit = "banana";  
char letter = fruit.charAt(0);
```

The argument `0` means that we want the character at **index** 0. String indexes range from 0 to $n - 1$, where n is the length of the string. So the character assigned to `letter` is 'b':

b	a	n	a	n	a
0	1	2	3	4	5

Characters work like the other data types you have seen. You can compare them using relational operators:

```
if (letter == 'A') {  
    System.out.println("It's an A!");  
}
```

Character literals, like 'A', appear in single quotes. Unlike string literals, which appear in double quotes, character literals can contain only a single character. Escape sequences, like '\t', are legal because they represent a single character.

The increment and decrement operators also work with characters. So this loop displays the letters of the alphabet:

```
System.out.print("Roman alphabet: ");  
for (char c = 'A'; c <= 'Z'; c++) {  
    System.out.print(c);  
}  
System.out.println();
```

The output is shown here:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Java uses **Unicode** to represent characters, so strings can store text in other alphabets like Cyrillic and Greek, and nonalphabetic languages like Chinese. You can read more about it at the [Unicode website](#).

In Unicode, each character is represented by a *code point*, which you can think of as an integer. The code points for uppercase Greek letters run from 913 to 937, so we can display the Greek alphabet like this:

```
System.out.print("Greek alphabet: ");
for (int i = 913; i <= 937; i++) {
    System.out.print((char) i);
}
System.out.println();
```

This example uses a type cast to convert each integer (in the range) to the corresponding character. Try running the code and see what happens.

Which Loop to Use

for and while loops have the same capabilities; any for loop can be rewritten as a while loop, and vice versa. For example, we could have printed letters of the alphabet by using a while loop:

```
System.out.print("Roman alphabet: ");
char c = 'A';
while (c <= 'Z') {
    System.out.print(c);
    c++;
}
System.out.println();
```

You might wonder when to use one or the other. It depends on whether you know how many times the loop will repeat.

A for loop is *definite*, which means we know, at the beginning of the loop, how many times it will repeat. In the alphabet example, we know it will run 26 times. In that case, it's better to use a for loop, which puts all of the loop control code on one line.

A while loop is *indefinite*, which means we don't know how many times it will repeat. For example, when validating user input as in [“Validating Input” on page 71](#), it's impossible to know how many times the user will enter a wrong value. In this case, a while loop is more appropriate:

```
System.out.print("Enter a number: ");
while (!in.hasNextDouble()) {
    String word = in.next();
    System.err.println(word + " is not a number");
    System.out.print("Enter a number: ");
}
double number = in.nextDouble();
```

It's easier to read the Scanner method calls when they're not all on one line of code.

String Iteration

Strings provide a method called `length` that returns the number of characters in the string. The following loop iterates the characters in `fruit` and displays them, one on each line:

```
for (int i = 0; i < fruit.length(); i++) {
    char letter = fruit.charAt(i);
    System.out.println(letter);
}
```

Because `length` is a method, you have to invoke it with parentheses (there are no arguments). When `i` is equal to the length of the string, the condition becomes `false` and the loop terminates.

To find the last letter of a string, you might be tempted to do something like this:

```
int length = fruit.length();
char last = fruit.charAt(length);    // wrong!
```

This code compiles and runs, but invoking the `charAt` method throws a `StringIndexOutOfBoundsException`. The problem is that there is no sixth letter in "banana". Since we started counting at 0, the six letters are indexed from 0 to 5. To get the last character, you have to subtract 1 from `length`:

```
int length = fruit.length();
char last = fruit.charAt(length - 1); // correct
```

Many string algorithms involve reading one string and building another. For example, to reverse a string, we can concatenate one character at a time:

```
public static String reverse(String s) {
    String r = "";
    for (int i = s.length() - 1; i >= 0; i--) {
        r += s.charAt(i);
    }
    return r;
}
```

The initial value of `r` is "", which is an **empty string**. The loop iterates the letters of `s` in reverse order. Each time through the loop, the `+=` operator creates a new string and assigns it to `r`. When the loop exits, `r` contains the letters from `s` in reverse order. So the result of `reverse("banana")` is "ananab".

The indexOf Method

To search for a specific character in a string, you could write a for loop and use `charAt` as in the previous section. However, the `String` class already provides a method for doing just that:

```
String fruit = "banana";  
int index = fruit.indexOf('a');    // returns 1
```

This example finds the index of 'a' in the string. But the letter appears three times, so it's not obvious what `indexOf` might do. According to the documentation, it returns the index of the *first* appearance.

To find subsequent appearances, you can use another version of `indexOf`, which takes a second argument that indicates where in the string to start looking:

```
int index = fruit.indexOf('a', 2); // returns 3
```

To visualize how `indexOf` and other `String` methods work, it helps to draw a picture like [Figure 6-3](#). The previous code starts at index 2 (the first 'n') and finds the next 'a', which is at index 3.

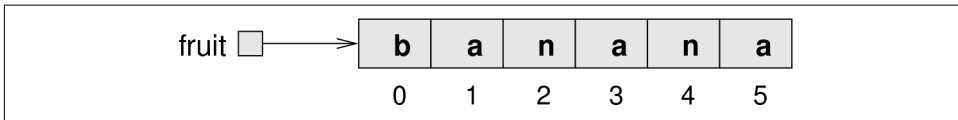


Figure 6-3. Memory diagram for a `String` of six characters

If the character happens to appear at the starting index, the starting index is the answer. So `fruit.indexOf('a', 5)` returns 5. If the character does not appear in the string, `indexOf` returns -1. Since indexes cannot be negative, this value indicates the character was not found.

You can also use `indexOf` to search for an entire string, not just a single character. For example, the expression `fruit.indexOf("nan")` returns 2.

Substrings

In addition to searching strings, we often need to extract parts of strings. The `substring` method returns a new string that copies letters from an existing string, given a pair of indexes:

- `fruit.substring(0, 3)` returns "ban"
- `fruit.substring(2, 5)` returns "nan"
- `fruit.substring(6, 6)` returns ""

Notice that the character indicated by the second index is *not* included. Defining `substring` this way simplifies some common operations. For example, to select a substring with length `len`, starting at index `i`, you could write `fruit.substring(i, i + len)`.

Like most string methods, `substring` is **overloaded**. That is, there are other versions of `substring` that have different parameters. If it's invoked with one argument, it returns the letters from that index to the end:

- `fruit.substring(0)` returns "banana"
- `fruit.substring(2)` returns "nana"
- `fruit.substring(6)` returns ""

The first example returns a copy of the entire string. The second example returns all but the first two characters. As the last example shows, `substring` returns the empty string if the argument is the length of the string.

We could also use `fruit.substring(2, fruit.length() - 1)` to get the result "nana". But calling `substring` with one argument is more convenient when you want the end of the string.

String Comparison

When comparing strings, it might be tempting to use the `==` and `!=` operators. But that will almost never work. The following code compiles and runs, but it always prints Goodbye! regardless what the user types.

```
System.out.print("Play again? ");
String answer = in.nextLine();
if (answer == "yes") { // wrong!
    System.out.println("Let's go!");
} else {
    System.out.println("Goodbye!");
}
```

The problem is that the `==` operator checks whether the two operands refer to the *same object*. Even if the answer is "yes", it will refer to a different object in memory than the literal string "yes" in the code. You'll learn more about objects and references in the next chapter.

The correct way to compare strings is with the `equals` method, like this:

```
if (answer.equals("yes")) {
    System.out.println("Let's go!");
}
```

This example invokes `equals` on `answer` and passes "yes" as an argument. The `equals` method returns true if the strings contain the same characters; otherwise, it returns false.

If two strings differ, we can use `compareTo` to see which comes first in alphabetical order:

```
String name1 = "Alan Turing";
String name2 = "Ada Lovelace";
int diff = name1.compareTo(name2);
if (diff < 0) {
    System.out.println("name1 comes before name2.");
} else if (diff > 0) {
    System.out.println("name2 comes before name1.");
} else {
    System.out.println("The names are the same.");
}
```

The return value from `compareTo` is the difference between the first characters in the strings that are not the same. In the preceding code, `compareTo` returns positive 8, because the second letter of "Ada" comes before the second letter of "Alan" by eight letters. If the first string (the one on which the method is invoked) comes earlier in the alphabet, the difference is negative. If it comes later in the alphabet, the difference is positive. If the strings are equal, their difference is zero.

Both `equals` and `compareTo` are case-sensitive. In Unicode, uppercase letters come before lowercase letters. So "Ada" comes before "ada".

String Formatting

In “[Formatting Output](#)” on page 34, we learned how to use `System.out.printf` to display formatted output. Sometimes programs need to create strings that are formatted a certain way, but not display them immediately (or ever). For example, the following method returns a time string in 12-hour format:

```
public static String timeString(int hour, int minute) {
    String ampm;
    if (hour < 12) {
        ampm = "AM";
        if (hour == 0) {
            hour = 12; // midnight
        }
    } else {
        ampm = "PM";
        hour = hour - 12;
    }
    return String.format("%02d:%02d %s", hour, minute, ampm);
}
```

`String.format` takes the same arguments as `System.out.printf`: a format specifier followed by a sequence of values. The main difference is that `System.out.printf` displays the result on the screen. `String.format` creates a new string but does not display anything.

In this example, the format specifier `%02d` means *two-digit integer padded with zeros*, so `timeString(19, 5)` returns the string `"07:05 PM"`. As an exercise, try writing two nested for loops (in `main`) that invoke `timeString` and display all possible times over a 24-hour period.

Be sure to skim through the documentation for `String`. Knowing what other methods are there will help you avoid reinventing the wheel. The easiest way to find documentation for Java classes is to do a web search for “Java” and the name of the class.

Vocabulary

loop

A statement that executes a sequence of statements repeatedly.

loop body

The statements inside the loop.

infinite loop

A loop whose condition is always true.

increment

Increase the value of a variable.

decrement

Decrease the value of a variable.

iteration

Executing a sequence of statements repeatedly.

loop variable

A variable that is initialized, tested, and updated in order to control a loop.

index

An integer variable or value used to indicate a character in a string.

Unicode

An international standard for representing characters in most of the world’s languages.

empty string

The string `" "`, which contains no characters and has a length of zero.

overloading

Defining two or more methods with the same name but different parameters.

Exercises

The code for this chapter is in the *ch06* directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

If you have not already read “Tracing with a Debugger” on page 265, now might be a good time. It describes the DrJava debugger, which is a useful tool for visualizing the flow of execution through loops.

Exercise 6-1.

Consider the following methods (`main` and `loop`):

```
public static void main(String[] args) {  
    loop(10);  
}
```

```
public static void loop(int n) {  
    int i = n;  
    while (i > 1) {  
        System.out.println(i);  
        if (i % 2 == 0) {  
            i = i / 2;  
        } else {  
            i = i + 1;  
        }  
    }  
}
```

1. Draw a table that shows the value of the variables `i` and `n` during the execution of `loop`. The table should contain one column for each variable and one line for each iteration.
2. What is the output of this program?
3. Can you prove that this loop terminates for any positive value of `n`?

Exercise 6-2.

Let's say you are given a number, a , and you want to find its square root. One way to do that is to start with a rough guess about the answer, x_0 , and then improve the guess by using this formula:

$$x_1 = (x_0 + a/x_0)/2$$

For example, if we want to find the square root of 9, and we start with $x_0 = 6$, then $x_1 = (6 + 9/6)/2 = 3.75$, which is closer. We can repeat the procedure, using x_1 to calculate x_2 , and so on. In this case, $x_2 = 3.075$ and $x_3 = 3.00091$. So the repetition converges quickly on the correct answer.

Write a method called `squareRoot` that takes a `double` and returns an approximation of the square root of the parameter, using this technique. You should not use `Math.sqrt`.

As your initial guess, you should use $a/2$. Your method should iterate until it gets two consecutive estimates that differ by less than 0.0001. You can use `Math.abs` to calculate the absolute value of the difference.

Exercise 6-3.

One way to evaluate $\exp(-x^2)$ is to use the infinite series expansion:

$$\exp(-x^2) = 1 - x^2 + x^4/2 - x^6/6 + \dots$$

The i th term in this series is $(-1)^i x^{2i}/i!$. Write a method named `gauss` that takes x and n as arguments and returns the sum of the first n terms of the series. You should not use `factorial` or `pow`.

Exercise 6-4.

A word is said to be *abecedarian* if the letters in the word appear in alphabetical order. For example, the following are all six-letter English abecedarian words:

abdest, acknow, acorsy, adempt, adipsy, agnosy, befist, behint, beknow, bijoux, biopsy, cestuy, chintz, deflux, dehors, dehort, deinos, diluvy, dimpsy

Write a method called `isAbecedarian` that takes a `String` and returns a `boolean` indicating whether the word is abecedarian.

Exercise 6-5.

A word is said to be a *doubloon* if every letter that appears in the word appears exactly twice. Here are some example doubloons found in the dictionary:

Abba, Anna, appall, appearer, appeases, arrainging, beriberi, bilabial, boob, Caucasus, coco, Dada, deed, Emmett, Hannah, horseshoer, intestines, Isis, mama, Mimi, murmur, noon, Otto, papa, peep, reappear, redder, sees, Shanghaiings, Toto

Write a method called `isDoubloon` that takes a string and checks whether it is a doubloon. To ignore case, invoke the `toLowerCase` method before checking.

Exercise 6-6.

In Scrabble each player has a set of tiles with letters on them. The object of the game is to use those letters to spell words. The scoring system is complex, but longer words are usually worth more than shorter words.

Imagine you are given your set of tiles as a string, like "quijibo", and you are given another string to test, like "jib".

Write a method called `canSpell` that takes two strings and checks whether the set of tiles can spell the word. You might have more than one tile with the same letter, but you can use each tile only once.

Arrays and References

Up to this point, the only variables we have used were for individual values such as numbers or strings. In this chapter, you'll learn how to store multiple values of the same type by using a single variable. This language feature will enable you to write programs that manipulate larger amounts of data.

For example, [Exercise 6-5](#) asked you to check whether every letter in a string appears exactly twice. One algorithm (which hopefully you already discovered) loops through the string 26 times, once for each lowercase letter:

```
// outer loop: for each lowercase letter
for (char c = 'a'; c <= 'z'; c++) {
    // inner loop: count how many times the letter appears
    for (int i = 0; i < str.length(); i++) {
        ...
    }
    // if the count is not 0 or 2, return false
}
```

This *nested loops* approach is inefficient, especially when the string is long. For example, there are more than 3 million characters in *War and Peace*; to process the whole book, the nested loop would run about 80 million times.

Another algorithm would initialize 26 variables to zero, loop through the string *one time*, and use a giant `if` statement to update the variable for each letter. But who wants to declare 26 variables?

That's where arrays come in. We can use a single variable to store 26 integers. Rather than use an `if` statement to update each value, we can use arithmetic to update the *n*th value directly. We will present this algorithm at the end of the chapter.

Creating Arrays

An **array** is a sequence of values; the values in the array are called **elements**. You can make an array of `ints`, `doubles`, `Strings`, or any other type, but all the values in an array must have the same type.

To create an array, you have to declare a variable with an *array type* and then create the array itself. Array types look like other Java types, except they are followed by square brackets (`[]`). For example, the following lines declare that `counts` is an *integer array* and `values` is a *double array*:

```
int[] counts;  
double[] values;
```

To create the array itself, you have to use the `new` operator, which you first saw in “The Scanner Class” on page 30. The `new` operator **allocates** memory for the array and automatically initializes all of its elements to zero:

```
counts = new int[4];  
values = new double[size];
```

The first assignment makes `counts` refer to an array of four integers. The second makes `values` refer to an array of `doubles`, but the number of elements depends on the value of `size` (at the time the array is created).

Of course, you can also declare the variable and create the array with a single line of code:

```
int[] counts = new int[4];  
double[] values = new double[size];
```

You can use any integer expression for the size of an array, as long as the value is non-negative. If you try to create an array with `-4` elements, for example, you will get a `NegativeArraySizeException`. An array with zero elements is allowed, and there are special uses for such arrays.

You can initialize an array with a comma-separated sequence of elements enclosed in braces, like this:

```
int[] a = {1, 2, 3, 4};
```

This statement creates an array variable, `a`, and makes it refer to an array with four elements.

Accessing Elements

When you create an array with the new operator, the elements are initialized to zero.

Figure 7-1 shows a memory diagram of the counts array so far.

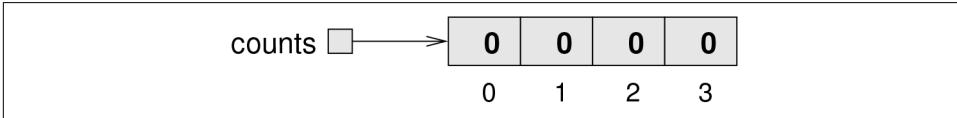


Figure 7-1. Memory diagram of an int array

The arrow indicates that the value of counts is a **reference** to the array. You should think of *the array* and *the variable* that refers to it as two different things. As you'll soon see, we can assign a different variable to refer to the same array, and we can change the value of counts to refer to a different array.

The boldface numbers inside the boxes are the elements of the array. The lighter numbers outside the boxes are the **indexes** used to identify each location in the array. As with strings, the index of the first element is 0, not 1. For this reason, we sometimes refer to the first element as the *zeroth* element.

The [] operator selects elements from an array:

```
System.out.println("The zeroth element is " + counts[0]);
```

You can use the [] operator anywhere in an expression:

```
counts[0] = 7;  
counts[1] = counts[0] * 2;  
counts[2]++;  
counts[3] -= 60;
```

Figure 7-2 shows the result of these statements.

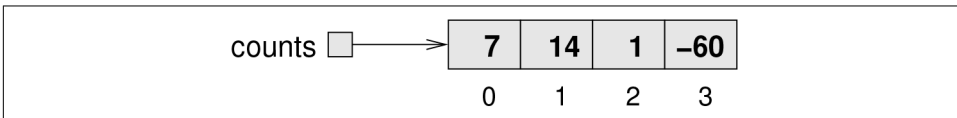


Figure 7-2. Memory diagram after several assignment statements

You can use any expression as an index, as long as it has type int. One of the most common ways to index an array is with a loop variable. For example:

```
int i = 0;  
while (i < 4) {  
    System.out.println(counts[i]);  
    i++;  
}
```

This `while` loop counts up from 0 to 4. When `i` is 4, the condition fails and the loop terminates. So the body of the loop is executed only when `i` is 0, 1, 2, or 3. In this context, the variable name `i` is short for *index*.

Each time through the loop, we use `i` as an index into the array, displaying the *i*th element. This type of array processing is usually written as a `for` loop:

```
for (int i = 0; i < 4; i++) {
    System.out.println(counts[i]);
}
```

For the `counts` array, the only legal indexes are 0, 1, 2, and 3. If the index is negative or greater than 3, the result is an `ArrayIndexOutOfBoundsException`.

Displaying Arrays

You can use `println` to display an array, but it probably doesn't do what you would like. For example, say you print an array like this:

```
int[] a = {1, 2, 3, 4};
System.out.println(a);
```

The output is something like this:

```
[I@bf3f7e0
```

The bracket indicates that the value is an array, `I` stands for *integer*, and the rest represents the address of the array in memory.

If we want to display the elements of the array, we can do it ourselves:

```
public static void printArray(int[] a) {
    System.out.print("{ " + a[0]);
    for (int i = 1; i < a.length; i++) {
        System.out.print(", " + a[i]);
    }
    System.out.println("}");
}
```

Given the previous array, the output of `printArray` is as follows:

```
{1, 2, 3, 4}
```

The Java library includes a class, `java.util.Arrays`, that provides methods for working with arrays. One of them, `toString`, returns a string representation of an array. After importing `Arrays`, we can invoke `toString` like this:

```
System.out.println(Arrays.toString(a));
```

And the output is shown here:

```
[1, 2, 3, 4]
```

Notice that `Arrays.toString` uses square brackets instead of curly braces. But it beats writing your own `printArray` method.

Copying Arrays

As explained in “[Accessing Elements](#)” on page 97, array variables contain *references* to arrays. When you make an assignment to an array variable, it simply copies the reference. But it doesn’t copy the array itself. For example:

```
double[] a = new double[3];
double[] b = a;
```

These statements create an array of three doubles and make two different variables refer to it, as shown in [Figure 7-3](#).

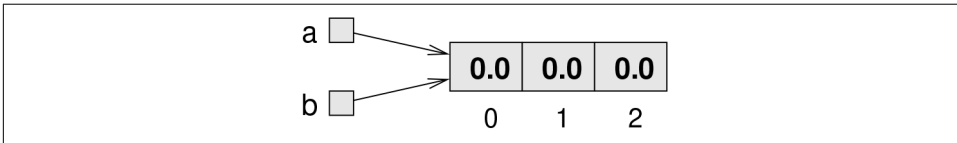


Figure 7-3. Memory diagram of two variables referring to the same array

Any changes made through either variable will be seen by the other. For example, if we set `a[0] = 17.0`, and then display `b[0]`, the result is `17.0`. Because `a` and `b` are different names for the same thing, they are sometimes called **aliases**.

If you actually want to copy the array, not just the reference, you have to create a new array and copy the elements from one to the other, like this:

```
double[] b = new double[3];
for (int i = 0; i < 3; i++) {
    b[i] = a[i];
}
```

`java.util.Arrays` provides a method named `copyOf` that performs this task for you. So you can replace the previous code with one line:

```
double[] b = Arrays.copyOf(a, 3);
```

The second parameter is the number of elements you want to copy, so `copyOf` can also be used to copy part of an array. [Figure 7-4](#) shows the state of the array variables after invoking `Arrays.copyOf`.

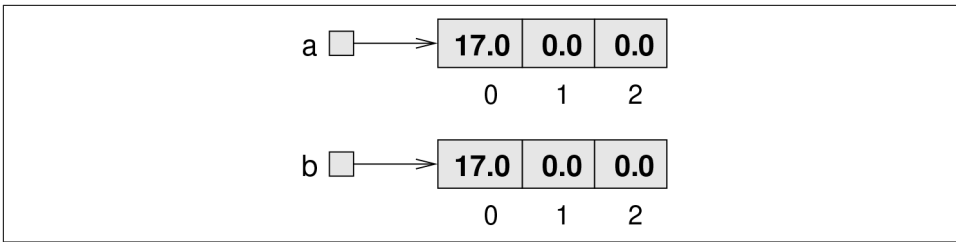


Figure 7-4. Memory diagram of two variables referring to different arrays

The examples so far work only if the array has three elements. It is better to generalize the code to work with arrays of any size. We can do that by replacing the magic number, 3, with `a.length`:

```
double[] b = new double[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

All arrays have a built-in constant, `length`, that stores the number of elements. In contrast to `String.length()`, which is a method, `a.length` is a constant. The expression `a.length` may look like a method invocation, but there are no parentheses and no arguments.

The last time the loop gets executed, `i` is `a.length - 1`, which is the index of the last element. When `i` is equal to `a.length`, the condition fails and the body is not executed—which is a good thing, because trying to access `a[a.length]` would throw an exception.

Of course, we can replace the loop altogether by using `Arrays.copyOf` and `a.length` for the second argument. The following line produces the same result shown in Figure 7-4:

```
double[] b = Arrays.copyOf(a, a.length);
```

The `Arrays` class provides many other useful methods like `Arrays.compare`, `Arrays.equals`, `Arrays.fill`, and `Arrays.sort`. Take a moment to read the documentation by searching the web for `java.util.Arrays`.

Traversing Arrays

Many computations can be implemented by looping through the elements of an array and performing an operation on each element. Looping through the elements of an array is called a **traversal**:

```
int[] a = {1, 2, 3, 4, 5};
for (int i = 0; i < a.length; i++) {
    a[i] *= a[i];
}
```

This example traverses an array and squares each element. At the end of the loop, the array has the values {1, 4, 9, 16, 25}.

Another common pattern is a **search**, which involves traversing an array and *searching* for a particular element. For example, the following method takes an array and a value, and it returns the index where the value appears:

```
public static int search(double[] array, double target) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == target) {
            return i;
        }
    }
    return -1; // not found
}
```

If we find the target value in the array, we return its index immediately. If the loop exits without finding the target, it returns -1, a special value chosen to indicate a failed search. (This code is essentially what the `String.indexOf` method does.)

The following code searches an array for the value 1.23, which is the third element. Because array indexes start at 0, the output is 2:

```
double[] array = {3.14, -55.0, 1.23, -0.8};
int index = search(array, 1.23);
System.out.println(index);
```

Another common traversal is a **reduce** operation, which *reduces* an array of values down to a single value. Examples include the sum or product of the elements, the minimum, and the maximum. The following method takes an array and returns the sum of its elements:

```
public static double sum(double[] array) {
    double total = 0.0;
    for (int i = 0; i < array.length; i++) {
        total += array[i];
    }
    return total;
}
```

Before the loop, we initialize `total` to 0. Each time through the loop, we update `total` by adding one element from the array. At the end of the loop, `total` contains the sum of the elements. A variable used this way is sometimes called an **accumulator**, because it *accumulates* the running total.

Generating Random Numbers

Most computer programs do the same thing every time they run; programs like that are called **deterministic**. Usually, determinism is a good thing, since we expect the same calculation to yield the same result. But for some applications, we want the computer to be unpredictable. Games are an obvious example, but there are many others, like scientific simulations.

Making a program **nondeterministic** turns out to be hard, because it's impossible for a computer to generate truly random numbers. But there are algorithms that generate unpredictable sequences called **pseudorandom** numbers. For most applications, they are as good as random.

If you did [Exercise 3-4](#), you have already seen `java.util.Random`, which generates pseudorandom numbers. The method `nextInt` takes an integer argument, `n`, and returns a random integer between 0 and `n - 1` (inclusive).

If you generate a long series of random numbers, every value should appear, at least approximately, the same number of times. One way to test this behavior of `nextInt` is to generate a large number of values, store them in an array, and count the number of times each value occurs.

The following method creates an `int` array and fills it with random numbers between 0 and 99. The argument specifies the desired size of the array, and the return value is a reference to the new array:

```
public static int[] randomArray(int size) {
    Random random = new Random();
    int[] a = new int[size];
    for (int i = 0; i < a.length; i++) {
        a[i] = random.nextInt(100);
    }
    return a;
}
```

The following `main` method generates an array and displays it by using `printArray` from [“Displaying Arrays” on page 98](#). We could have used `Arrays.toString`, but we like seeing curly braces instead of square brackets:

```
public static void main(String[] args) {
    int[] array = randomArray(8);
    printArray(array);
}
```

Each time you run the program, you should get different values. The output will look something like this:

```
{15, 62, 46, 74, 67, 52, 51, 10}
```


Building a Histogram

If these values were exam scores—and they would be pretty bad exam scores in that case—the teacher might present them to the class in the form of a **histogram**. In statistics, a histogram is a set of counters that keeps track of the number of times each value appears.

For exam scores, we might have 10 counters to keep track of how many students scored in the 90s, the 80s, etc. To do that, we can traverse the array and count the number of elements that fall in a given range.

The following method takes an array and two integers. It returns the number of elements that fall in the range from `low` to `high - 1`:

```
public static int inRange(int[] a, int low, int high) {
    int count = 0;
    for (int i = 0; i < a.length; i++) {
        if (a[i] >= low && a[i] < high) {
            count++;
        }
    }
    return count;
}
```

This pattern should look familiar: it is another reduce operation. Notice that `low` is included in the range (`>=`), but `high` is excluded (`<`). This design keeps us from counting any scores twice.

Now we can count the number of scores in each grade range. We add the following code to our `main` method:

```
int[] scores = randomArray(30);
int a = inRange(scores, 90, 100);
int b = inRange(scores, 80, 90);
int c = inRange(scores, 70, 80);
int d = inRange(scores, 60, 70);
int f = inRange(scores, 0, 60);
```

This code is repetitive, but it is acceptable as long as the number of ranges is small. Suppose we wanted to keep track of the number of times each individual score appears. Then we would have to write 100 lines of code:

```
int count0 = inRange(scores, 0, 1);
int count1 = inRange(scores, 1, 2);
int count2 = inRange(scores, 2, 3);
...
int count99 = inRange(scores, 99, 100);
```

What we need is a way to store 100 counters, preferably so we can use an index to access them. Wait a minute—that's exactly what an array does.

The following fragment creates an array of 100 counters, one for each possible score. It loops through the scores and uses `inRange` to count how many times each score appears. Then it stores the results in the `counts` array:

```
int[] counts = new int[100];
for (int i = 0; i < counts.length; i++) {
    counts[i] = inRange(scores, i, i + 1);
}
```

Notice that we are using the loop variable `i` three times: as an index into the `counts` array, and in the last two arguments of `inRange`.

The code works, but it is not as efficient as it could be. Every time the loop invokes `inRange`, it traverses the entire array. It would be better to make a single pass through the `scores` array.

For each score, we already know which range it falls in—the score itself. We can use that value to increment the corresponding counter. This code traverses the array of scores *only once* to generate the histogram:

```
int[] counts = new int[100];
for (int i = 0; i < scores.length; i++) {
    int index = scores[i];
    counts[index]++;
}
```

Each time through the loop, it selects one element from `scores` and uses it as an index to increment the corresponding element of `counts`. Because this code traverses the array of scores only once, it is much more efficient.

The Enhanced for Loop

Since traversing arrays is so common, Java provides an alternative syntax that makes the code more compact. Consider a `for` loop that displays the elements of an array on separate lines:

```
for (int i = 0; i < values.length; i++) {
    int value = values[i];
    System.out.println(value);
}
```

We could rewrite the loop like this:

```
for (int value : values) {
    System.out.println(value);
}
```

This statement is called an **enhanced for loop**, also known as the *for each* loop. You can read the code as, “for each `value` in `values`”. It’s conventional to use plural nouns for array variables and singular nouns for element variables.

Notice how the single line `for (int value : values)` replaces the first two lines of the standard `for` loop. It hides the details of iterating each index of the array, and instead, focuses on the values themselves.

Using the enhanced `for` loop, and removing the temporary variable, we can write the histogram code from the previous section more concisely:

```
int[] counts = new int[100];
for (int score : scores) {
    counts[score]++;
}
```

Enhanced `for` loops often make the code more readable, especially for accumulating values. But they are not helpful when you need to refer to the index, as in search operations:

```
for (double d : array) {
    if (d == target) {
        // array contains d, but we don't know where
    }
}
```

Counting Characters

We now return to the example from the beginning of the chapter and present a solution to [Exercise 6-5](#) using arrays. Here is the problem again:

A word is said to be a *doubloon* if every letter that appears in the word appears exactly twice. Write a method called `isDoubloon` that takes a string and checks whether it is a doubloon. To ignore case, invoke the `toLowerCase` method before checking.

Based on the approach from [“Building a Histogram” on page 103](#), we will create an array of 26 integers to count how many times each letter appears. We convert the string to lowercase, so that we can treat "A" and "a" (for example) as the same letter:

```
int[] counts = new int[26];
String lower = s.toLowerCase();
```

We can use a `for` loop to iterate each character in the string. To update the counts array, we need to compute the index that corresponds to each character. Fortunately, Java allows you to perform arithmetic on characters:

```
for (int i = 0; i < lower.length(); i++) {
    char letter = lower.charAt(i);
    int index = letter - 'a';
    counts[index]++;
}
```

If `letter` is "a", the value of `index` is 0; if `letter` is "b", the value of `index` is 1, and so on.

Then we use `index` to increment the corresponding element of `counts`. At the end of the loop, `counts` contains a histogram of the letters in the string `lower`.

We can simplify this code with an enhanced for loop, but it doesn't work with strings; we have to convert `lower` to an array of characters, like this:

```
for (char letter : lower.toCharArray()) {
    int index = letter - 'a';
    counts[index]++;
}
```

Once we have the counts, we can use a second for loop to check whether each letter appears zero or two times:

```
for (int count : counts) {
    if (count != 0 && count != 2) {
        return false; // not a doubleton
    }
}
return true; // is a doubleton
```

If we find a count that is neither 0 or 2, we know the word is not a doubleton and we can return immediately. If we make it all the way through the for loop, we know that all counts are 0 or 2, which means the word is a doubleton.

Pulling together the code fragments, and adding some comments and test cases, here's the entire program:

```
public class Doubleton {

    public static boolean isDoubleton(String s) {
        // count the number of times each letter appears
        int[] counts = new int[26];
        String lower = s.toLowerCase();
        for (char letter : lower.toCharArray()) {
            int index = letter - 'a';
            counts[index]++;
        }
        // determine whether the given word is a doubleton
        for (int count : counts) {
            if (count != 0 && count != 2) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        System.out.println(isDoubleton("Mama")); // true
        System.out.println(isDoubleton("Lama")); // false
    }
}
```

This example uses methods, `if` statements, `for` loops, arithmetic and logical operators, integers, characters, strings, booleans, and arrays. We hope you'll take a second to appreciate how much you've learned!

Vocabulary

array

A collection of values in which all the values have the same type, and each value is identified by an index.

element

One of the values in an array. The `[]` operator selects elements.

allocate

To reserve memory for an array or other object. In Java, the `new` operator allocates memory.

reference

A value that indicates a storage location. In a memory diagram, a reference appears as an arrow.

index

An integer variable or value used to indicate an element of an array; see also [Chapter 6](#).

alias

A variable that refers to the same object as another variable.

traversal

Looping through the elements of an array (or other collection).

search

A traversal pattern used to find a particular element of an array.

reduce

A traversal pattern that combines the elements of an array into a single value.

accumulator

A variable used to accumulate results during a traversal.

deterministic

A program that does the same thing every time it is run.

nondeterministic

A program that always behaves differently, even when run multiple times with the same input.

pseudorandom

A sequence of numbers that appear to be random but are actually the product of a deterministic computation.

histogram

An array of integers in which each integer counts the number of values that fall into a certain range.

enhanced for loop

An alternative syntax for traversing the elements of an array (or other collection).

Exercises

The code for this chapter is in the *ch07* directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

If you haven’t already, take a look at [Appendix D](#), where we’ve collected some of our favorite debugging advice. It refers to language features we haven’t yet covered, but it’s good for you to know what’s available when you need it.

Exercise 7-1.

The purpose of this exercise is to practice reading code and recognizing the traversal patterns in this chapter. The following methods are hard to read, because instead of using meaningful names for the variables and methods, they use names of fruit.

For each method, write one sentence that describes what the method does, without getting into the details of how it works. And for each variable, identify the role it plays.

```
public static int banana(int[] a) {
    int kiwi = 1;
    int i = 0;
    while (i < a.length) {
        kiwi = kiwi * a[i];
        i++;
    }
    return kiwi;
}

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

```

public static int pineapple(int[] a, int apple) {
    int pear = 0;
    for (int pine: a) {
        if (pine == apple) {
            pear++;
        }
    }
    return pear;
}

```

Exercise 7-2.

What is the output of the following program? Describe in a few words what `mus` does. Draw a stack diagram just before `mus` returns.

```

public static int[] make(int n) {
    int[] a = new int[n];
    for (int i = 0; i < n; i++) {
        a[i] = i + 1;
    }
    return a;
}

public static void dub(int[] jub) {
    for (int i = 0; i < jub.length; i++) {
        jub[i] *= 2;
    }
}

public static int mus(int[] zoo) {
    int fus = 0;
    for (int i = 0; i < zoo.length; i++) {
        fus += zoo[i];
    }
    return fus;
}

public static void main(String[] args) {
    int[] bob = make(5);
    dub(bob);
    System.out.println(mus(bob));
}

```

Exercise 7-3.

Write a method called `indexOfMax` that takes an array of integers and returns the index of the largest element. Can you write this method by using an enhanced `for` loop? Why or why not?

Exercise 7-4.

The **Sieve of Eratosthenes** is “a simple, ancient algorithm for finding all prime numbers up to any given limit.”

Write a method called `sieve` that takes an integer parameter, `n`, and returns a `boolean` array that indicates, for each number from 0 to `n - 1`, whether the number is prime.

Exercise 7-5.

Write a method `areFactors` that takes an integer `n` and an array of integers, and returns `true` if the numbers in the array are all factors of `n` (which is to say that `n` is divisible by all of them).

Exercise 7-6.

Write a method named `arePrimeFactors` that takes an integer `n` and an array of integers, and that returns `true` if the numbers in the array are all prime *and* their product is `n`.

Exercise 7-7.

Write a method called `letterHist` that takes a string as a parameter and returns a histogram of the letters in the string. The zeroth element of the histogram should contain the number of a's in the string (upper- and lowercase); the 25th element should contain the number of z's. Your solution should traverse the string only once.

Exercise 7-8.

Two words are *anagrams* if they contain the same letters and the same number of each letter. For example, *stop* is an anagram of *pots*, *allen downey* is an anagram of *well annoyed*, and *christopher mayfield* is an anagram of *hi prof the camel is dry*. Write a method that takes two strings and checks whether they are anagrams of each other.

Recursive Methods

Up to this point, we've been using `while` and `for` loops whenever we've needed to repeat something. Methods that use iteration are called **iterative**. They are straightforward, but sometimes more-elegant solutions exist.

In this chapter, we explore one of the most magical things that a method can do: invoke *itself* to solve a smaller version of the *same* problem. A method that invokes itself is called **recursive**.

Recursive Void Methods

Consider the following example:

```
public static void countdown(int n) {
    if (n == 0) {
        System.out.println("Blastoff!");
    } else {
        System.out.println(n);
        countdown(n - 1);
    }
}
```

The name of the method is `countdown`; it takes a single integer as a parameter. If the parameter is 0, it displays the word *Blastoff*. Otherwise, it displays the number and then invokes itself, passing `n - 1` as the argument.

What happens if we invoke `countdown(3)` from `main`?

The execution of `countdown` begins with `n == 3`, and since `n` is not 0, it displays the value 3, and then invokes itself..

The execution of `countdown` begins with `n == 2`, and since `n` is not 0, it displays the value 2, and then invokes itself..

The execution of `countdown` begins with `n == 1`, and since `n` is not zero, it displays the value 1, and then invokes itself..

The execution of `countdown` begins with `n == 0`, and since `n` is 0, it displays the word *Blastoff!* and then returns.

The `countdown` that got `n == 1` returns.

The `countdown` that got `n == 2` returns.

The `countdown` that got `n == 3` returns.

And then you're back in `main`. So the total output looks like this:

```
3
2
1
Blastoff!
```

As a second example, we'll rewrite the methods `newLine` and `threeLine` from “[Defining New Methods](#)” on page 45. Here they are again:

```
public static void newLine() {
    System.out.println();
}

public static void threeLine() {
    newLine();
    newLine();
    newLine();
}
```

Although these methods work, they would not help if we wanted to display two newlines, or maybe 100. A more general alternative would be the following:

```
public static void nLines(int n) {
    if (n > 0) {
        System.out.println();
        nLines(n - 1);
    }
}
```

This method takes an integer, `n`, as a parameter and displays `n` newlines. The structure is similar to `countdown`. As long as `n` is greater than 0, it displays a newline and then invokes itself to display $(n - 1)$ additional newlines. The total number of newlines is $1 + (n - 1)$, which is just what we wanted: `n`.

Recursive Stack Diagrams

In “Stack Diagrams” on page 50, we used a stack diagram to represent the state of a program during a method invocation. The same kind of diagram can make it easier to interpret a recursive method.

Remember that every time a method gets called, Java creates a new frame that contains the method’s parameters and variables. Figure 8-1 is a stack diagram for `countdown`, called with `n == 3`.

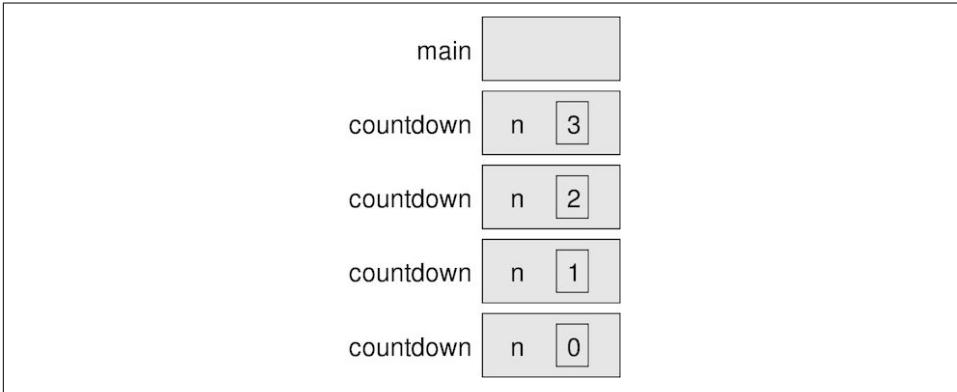


Figure 8-1. Stack diagram for the `countdown` program

By convention, the frame for `main` is at the top, and the stack of other frames grows down. That way, we can draw stack diagrams on paper without needing to guess how far they will grow. The frame for `main` is empty because `main` does not have any variables. (It has the parameter `args`, but since we’re not using it, we left it out of the diagram.)

There are four frames for `countdown`, each with a different value for the parameter `n`. The last frame, with `n == 0`, is called the **base case**. It does not make a recursive call, so there are no more frames below it.

If there is no base case in a recursive method, or if the base case is never reached, the stack would grow forever—at least in theory. In practice, the size of the stack is limited. If you exceed the limit, you get a `StackOverflowError`.

For example, here is a recursive method without a base case:

```
public static void forever(String s) {  
    System.out.println(s);  
    forever(s);  
}
```

This method displays the given string until the stack overflows, at which point it throws an error. Try this example on your computer—you might be surprised by how long the error message is!

Value-Returning Methods

To give you an idea of what you can do with the tools you have learned, let's look at methods that evaluate recursively defined mathematical functions.

A *recursive definition* is similar to a *circular definition*, in the sense that the definition refers to the thing being defined. Of course, a truly circular definition is not very useful:

recursive: An adjective used to describe a method that is recursive.

If you saw that definition in the dictionary, you might be annoyed. Then again, if you search for “recursion” on Google, it will ask, “Did you mean recursion?” as an inside joke. People fall for that link all the time.

Many mathematical functions are defined recursively, because that is often the simplest way. For example, the **factorial** of an integer n , which is written $n!$, is defined like this:

$$0! = 1$$
$$n! = n \cdot (n - 1)!$$

Don't confuse the mathematical symbol $!$, which means *factorial*, with the Java operator `!`, which means *not*. This definition says that `factorial(0)` is 1, and `factorial(n)` is $n * \text{factorial}(n - 1)$.

So `factorial(3)` is $3 * \text{factorial}(2)$; `factorial(2)` is $2 * \text{factorial}(1)$; `factorial(1)` is $1 * \text{factorial}(0)$; and `factorial(0)` is 1. Putting it all together, we get $3 * 2 * 1 * 1$, which is 6.

If you can formulate a recursive definition of something, you can easily write a Java method to evaluate it. The first step is to decide what the parameters and return type are. Since `factorial` is defined for integers, the method takes an `int` as a parameter and returns an `int`:

```
public static int factorial(int n) {  
    return 0; // stub  
}
```

Next, we think about the base case. If the argument happens to be 0, we return 1:

```

public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return 0; // stub
}

```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n - 1$, and then multiply it by n :

```

public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    int recurse = factorial(n - 1);
    int result = n * recurse;
    return result;
}

```

To illustrate what is happening, we'll use the temporary variables `recurse` and `result`. In each method call, `recurse` stores the factorial of $n - 1$, and `result` stores the factorial of n .

The flow of execution for this program is similar to countdown from “[Recursive Void Methods](#)” on page 111. If we invoke `factorial` with the value 3:

Since 3 is not 0, we skip the first branch and calculate the factorial of $n - 1$...

 Since 2 is not 0, we skip the first branch and calculate the factorial of $n - 1$...

 Since 1 is not 0, we skip the first branch and calculate the factorial of $n - 1$...

 Since 0 is 0, we take the first branch and return the value 1 immediately.

 The return value (1) gets multiplied by n , which is 1, and the result is returned.

 The return value (1) gets multiplied by n , which is 2, and the result is returned.

Since the return value (2) gets multiplied by n , which is 3, and the result, 6, is returned to whatever invoked `factorial(3)`.

Figure 8-2 shows what the stack diagram looks like for this sequence of method invocations. The return values are shown being passed up the stack. Notice that `recurse` and `result` do not exist in the last frame, because when `n == 0`, the code that declares them does not execute.

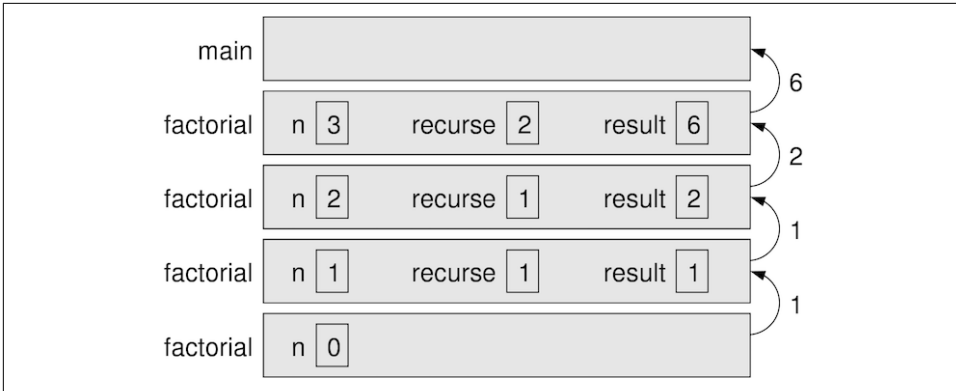


Figure 8-2. Stack diagram for the `factorial` method

The Leap of Faith

Following the flow of execution is one way to read programs, but it can quickly become overwhelming. Another way to understand recursion is the **leap of faith**: when you come to a method invocation, instead of following the flow of execution, you *assume* that the method works correctly and returns the appropriate value.

In fact, you are already practicing this leap of faith when you use methods in the Java library. When you invoke `Math.cos` or `System.out.println`, you don't think about the implementations of those methods. You just assume that they work properly.

The same is true of other methods. For example, consider the method from “[Boolean Methods](#)” on page 70 that determines whether an integer has only one digit:

```
public static boolean isSingleDigit(int x) {
    return x > -10 && x < 10;
}
```

Once you convince yourself that this method is correct—by examining and testing the code—you can just use the method without ever looking at the implementation again.

Recursive methods are no different. When you get to a recursive call, don't think about the flow of execution. Instead, *assume* that the recursive call produces the desired result.

For example, “Assuming that I can find the factorial of $n - 1$, can I compute the factorial of n ?” Yes you can, by multiplying by n . Here's an implementation of `factorial` with the temporary variables removed:

```

public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}

```

Notice how similar this version is to the original mathematical definition:

$$0! = 1$$

$$n! = n \cdot (n - 1)!$$

Of course, it is strange to assume that the method works correctly when you have not finished writing it. But that's why it's called a leap of faith!

Another common recursively defined mathematical function is the Fibonacci sequence, which has the following definition:

$$fibonacci(1) = 1$$

$$fibonacci(2) = 1$$

$$fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)$$

Notice that each Fibonacci number is the sum of the two preceding Fibonacci numbers. Translated into Java, this function is as follows:

```

public static int fibonacci(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

```

If you try to follow the flow of execution here, even for small values of n , your head will explode. But if we take a leap of faith and assume that the two recursive invocations work correctly, then it is clear, looking at the definition, that our implementation is correct.

Counting Up Recursively

The countdown example in “[Recursive Void Methods](#)” on page 111 has three parts: (1) it checks the base case, (2) it displays something, and (3) it makes a recursive call. What do you think happens if you reverse steps 2 and 3, making the recursive call *before* displaying?

```

public static void countup(int n) {
    if (n == 0) {
        System.out.println("Blastoff!");
    } else {
        countup(n - 1);
        System.out.println(n);
    }
}

```

The stack diagram is the same as before, and the method is still called n times. But now the `System.out.println` happens just before each recursive call returns. As a result, it counts *up* instead of down:

```

Blastoff!
1
2
3

```

Keep this in mind for the next example, which displays numbers in binary.

Binary Number System

You are probably aware that computers can store only 1s and 0s. That's because processors and memory are made up of billions of tiny on-off switches.

The value 1 means a switch is on; the value 0 means a switch is off. All types of data, whether integer, floating-point, text, audio, video, or something else, are represented by 1s and 0s.

Fortunately, we can represent any integer as a **binary** number. [Table 8-1](#) shows the first eight numbers in binary and decimal.

Table 8-1. The first eight binary numbers

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

In decimal there are 10 digits, and the written representation of numbers is based on powers of 10. For example, the number 456 has 4 in the 100's place, 5 in the 10's place, and 6 in the 1's place. So the value is $400 + 50 + 6$:

4	5	6
10^2	10^1	10^0

In binary there are two digits, and the written representation of numbers is based on powers of two. For example, the number 10111 has 1 in the 16's place, 0 in the 8's place, 1 in the 4's place, 1 in the 2's place, and 1 in the 1's place. So the value is $16 + 0 + 4 + 2 + 1$, which is 23 in decimal.

1	0	1	1	1
2^4	2^3	2^2	2^1	2^0

To get the digits of a decimal number, we can use repeated division. For example, if we divide 456 by 10, we get 45 with remainder 6. The remainder is the rightmost digit of 456.

If we divide the result again, we get 4 with remainder 5. The remainder is the second rightmost digit of 456. And if we divide again, we get 0 with remainder 4. The remainder is the third rightmost digit of 456, and the result, 0, tells us that we're done.

We can do the same thing in binary if we divide by 2. When you divide by 2, the remainder is the rightmost digit, either 0 or 1. If you divide the result again, you get the second rightmost digit. If you keep going, and write down the remainders, you'll have your number in binary:

```
23 / 2 is 11 remainder 1
11 / 2 is 5 remainder 1
5 / 2 is 2 remainder 1
2 / 2 is 1 remainder 0
1 / 2 is 0 remainder 1
```

Reading these remainders from bottom to top, 23 in binary is 10111.

Recursive Binary Method

Now, to display a number in binary, we can combine the algorithm from the previous section and the *count up* pattern from [“Counting Up Recursively” on page 117](#).

Here is a recursive method that displays any positive integer in binary:

```

public static void displayBinary(int value) {
    if (value > 0) {
        displayBinary(value / 2);
        System.out.print(value % 2);
    }
}

```

If `value` is 0, `displayBinary` does nothing (that's the base case). If the argument is positive, the method divides it by 2 and calls `displayBinary` recursively. When the recursive call returns, the method displays one digit of the result and returns (again). [Figure 8-3](#) illustrates this process.

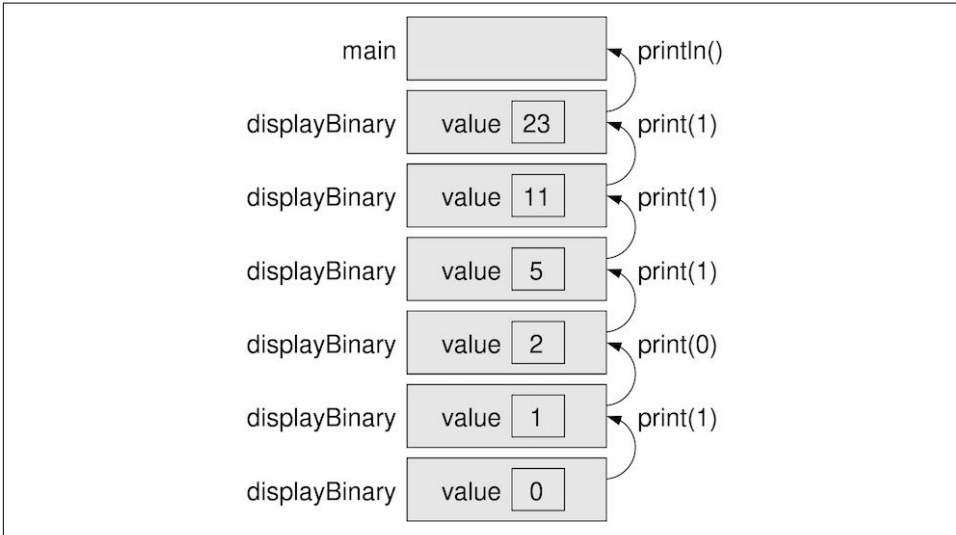


Figure 8-3. Stack diagram for the `displayBinary` method

The leftmost digit is near the bottom of the stack, so it gets displayed first. The rightmost digit, near the top of the stack, gets displayed last. After invoking `displayBinary`, we use `println` to complete the output:

```

displayBinary(23);    // output is 10111
System.out.println();

```

CodingBat Problems

In the past several chapters, you've seen methods, conditions, loops, strings, arrays, and recursion. A great resource for practicing all of these concepts is [CodingBat](#).

CodingBat is a free website of programming problems developed by Nick Parlante, a computer science lecturer at Stanford University. As you work on these problems, CodingBat saves your progress (if you create an account).

To conclude this chapter, we consider two problems in the Recursion-1 section of CodingBat. One of them deals with strings, and the other deals with arrays. Both of them have the same recursive idea: check the base case, look at the current index, and recursively handle the rest.

The first problem is available at <https://codingbat.com/prob/p118230>:

Recursion-1 noX

Given a string, compute recursively a new string where all the "x" chars have been removed.

`noX("xaxb") → "ab" noX("abc") → "abc" noX("xx") → ""`

When solving recursive problems, it helps to think about the base case first. The base case is the easiest version of the problem; for `noX`, it's the empty string. If the argument is an empty string, there are no x's to be removed:

```
if (str.length() == 0) {  
    return "";  
}
```

Next comes the more difficult part. To solve a problem recursively, you need to think of a simpler instance of the same problem. For `noX`, it's removing all the x's from a shorter string.

So let's split the string into two parts, the first letter and the rest:

```
char first = str.charAt(0);  
String rest = str.substring(1);
```

Now we can make a recursive call to remove the x's from rest:

```
String recurse = noX(rest);
```

If `first` happens to be an x, we're done; we just have to return `recurse`. Otherwise, we have to concatenate `first` and `recurse`. Here's the `if` statement we need:

```
if (first == 'x') {  
    return recurse;  
} else {  
    return first + recurse;  
}
```

You can run this solution on CodingBat by pasting these snippets into the provided method definition.

The second problem is available at <https://codingbat.com/prob/p135988>:

Recursion-1 array11

Given an array of ints, compute recursively the number of times that the value 11 appears in the array.

array11([1, 2, 11], 0) → 1 array11([11, 11], 0) → 2 array11([1, 2, 3, 4], 0) → 0

This problem uses the convention of passing the index as an argument. So the base case is when we've reached the end of the array. At that point, we know there are no more 11s:

```
if (index >= nums.length) {
    return 0;
}
```

Next we look at the current number (based on the given index), and check if it's an 11. After that, we can recursively check the rest of the array. Similar to the noX problem, we look at only one integer per method call:

```
int recurse = array11(nums, index + 1);
if (nums[index] == 11) {
    return recurse + 1;
} else {
    return recurse;
}
```

Again, you can run this solutions on CodingBat by pasting the snippets into the method definition.

To see how these solutions actually work, you might find it helpful to step through them with a debugger (see “[Tracing with a Debugger](#)” on page 265) or [Java Tutor](#). Then try solving other CodingBat problems on your own.

Learning to think recursively is an important part of learning to think like a computer scientist. Many algorithms can be written concisely with recursive methods that perform computations on the way down, on the way up, or both.

Vocabulary

iterative

A method or algorithm that repeats steps by using one or more loops.

recursive

A method or algorithm that invokes itself one or more times with different arguments.

base case

A condition that causes a recursive method *not* to make another recursive call.

factorial

The product of all the integers up to and including a given integer.

leap of faith

A way to read recursive programs by assuming that the recursive call works, rather than following the flow of execution.

binary

A system that uses only zeros and ones to represent numbers. Also known as *base 2*.

Exercises

The code for this chapter is in the *ch08* directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

If you have not already read “Testing with JUnit” on page 267, now might be a good time. It describes JUnit, a standard framework for writing test code.

Exercise 8-1.

The purpose of this exercise is to take a problem and break it into smaller problems, and to solve the smaller problems by writing simple methods. Consider the first verse of the song “99 Bottles of Beer”:

99 bottles of beer on the wall, 99 bottles of beer, ya’ take one down, ya’ pass it around,
98 bottles of beer on the wall.

Subsequent verses are identical except that the number of bottles gets smaller by one in each verse, until the last verse:

No bottles of beer on the wall, no bottles of beer, ya’ can’t take one down, ya’ can’t pass
it around, ’cause there are no more bottles of beer on the wall!

And then the song (finally) ends.

Write a program that displays the entire lyrics of “99 Bottles of Beer”. Your program should include a recursive method that does the hard part, but you might want to write additional methods to separate other parts of the program. As you develop your code, test it with a small number of verses, like 3.

Exercise 8-2.

Write a recursive method named `oddSum` that takes a positive odd integer n and returns the sum of odd integers from 1 to n . Start with a base case, and use temporary

variables to debug your solution. You might find it helpful to print the value of `n` each time `oddSum` is invoked.

Exercise 8-3.

In this exercise, you will use a stack diagram to understand the execution of the following recursive method:

```
public static void main(String[] args) {
    System.out.println(prod(1, 4));
}

public static int prod(int m, int n) {
    if (m == n) {
        return n;
    } else {
        int recurse = prod(m, n - 1);
        int result = n * recurse;
        return result;
    }
}
```

1. Draw a stack diagram showing the state of the program just before the last invocation of `prod` completes.
2. What is the output of this program? (Try to answer this question on paper first; then run the code to check your answer.)
3. Explain in a few words what `prod` does (without getting into the details of how it works).
4. Rewrite `prod` without the temporary variables `recurse` and `result`. *Hint:* You need only one line for the `else` branch.

Exercise 8-4.

The goal of this exercise is to translate a recursive definition into a Java method. The Ackermann function is defined for non-negative integers as follows:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Write a recursive method called `ack` that takes two `ints` as parameters and that computes and returns the value of the Ackermann function.

Test your implementation of Ackermann by invoking it from `main` and displaying the return value. Note the return value gets very big very quickly. You should try it only for small values of m and n (not bigger than 3).

Exercise 8-5.

Write a recursive method called `power` that takes a double x and an integer n and returns x^n .

Hint: A recursive definition of this operation is $x^n = x \cdot x^{n-1}$. Also, remember that anything raised to the zeroth power is 1.

Optional challenge: you can make this method more efficient, when n is even, using $x^n = (x^{n/2})^2$.

Exercise 8-6.

Many of the patterns you have seen for traversing arrays can also be written recursively. It is not common, but it is a useful exercise.

1. Write a method called `maxInRange` that takes an array of integers and two indexes, `lowIndex` and `highIndex`, and finds the maximum value in the array, but considering only the elements between `lowIndex` and `highIndex`, including both.

This method should be recursive. If the length of the range is 1 (i.e., if `lowIndex == highIndex`), we know immediately that the sole element in the range must be the maximum. So that's the base case.

If there is more than one element in the range, we can break the array into two pieces, find the maximum in each piece, and then find the maximum of the maxima.

2. Methods like `maxInRange` can be awkward to use. To find the largest element in an array, we have to provide the range for the entire array:

```
double max = maxInRange(a, 0, a.length - 1);
```

Write a method called `max` that takes an array and uses `maxInRange` to find and return the largest element.

Exercise 8-7.

Create a program called `Recurse.java` and type in the following methods:

```
/**
 * Returns the first character of the given String.
 */
public static char first(String s) {
    return s.charAt(0);
}

/**
 * Returns all but the first letter of the given String.
 */
public static String rest(String s) {
    return s.substring(1);
}

/**
 * Returns all but the first and last letter of the String.
 */
public static String middle(String s) {
    return s.substring(1, s.length() - 1);
}

/**
 * Returns the length of the given String.
 */
public static int length(String s) {
    return s.length();
}
```

1. Write some code in `main` that tests each of these methods. Make sure they work, and you understand what they do.
2. Using these methods, and without using any other `String` methods, write a method called `printString` that takes a string as a parameter and displays the letters of the string, one on each line. It should be a `void` method.
3. Again using only these methods, write a method called `printBackward` that does the same thing as `printString` but displays the string backward (again, one character per line).
4. Now write a method called `reverseString` that takes a string as a parameter and returns a new string as a return value. The new string should contain the same letters as the parameter, but in reverse order:

```
String backwards = reverseString("coffee");
System.out.println(backwards);
```


The output of this example code should be as follows:

```
eeffoc
```

5. A *palindrome* is a word that reads the same both forward and backward, like *otto* and *palindromeemordnilap*. Here's one way to test whether a string is a palindrome:

A single letter is a palindrome, a two-letter word is a palindrome if the letters are the same, and any other word is a palindrome if the first letter is the same as the last and the middle is a palindrome.

Write a recursive method named `isPalindrome` that takes a `String` and returns a `boolean` indicating whether the word is a palindrome.

Immutable Objects

Java is an **object-oriented** language, which means that it uses objects to (1) represent data and (2) provide methods related to them. This way of organizing programs is a powerful design concept, and we will introduce it gradually throughout the remainder of the book.

An **object** is a collection of data that provides a set of methods. For example, `Scanner`, which you saw in “[The Scanner Class](#)” on page 30, is an object that provides methods for parsing input. `System.out` and `System.in` are also objects.

Strings are objects too. They contain characters and provide methods for manipulating character data. Other data types, like `Integer`, contain numbers and provide methods for manipulating number data. We will explore some of these methods in this chapter.

Primitives Versus Objects

Not everything in Java is an object: `int`, `double`, `char`, and `boolean` are **primitive** types. When you declare a variable with a primitive type, Java reserves a small amount of memory to store its value. [Figure 9-1](#) shows how the following values are stored in memory:

```
int number = -2;  
char symbol = '!';
```

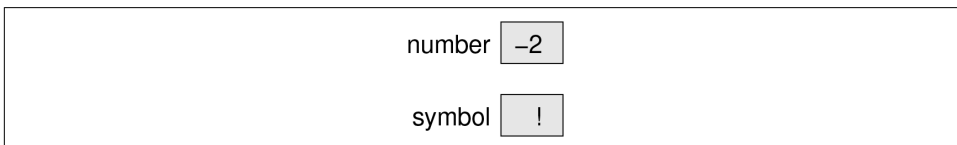


Figure 9-1. Memory diagram of two primitive variables

As you learned in “Accessing Elements” on page 97, an array variable stores a *reference* to an array. For example, the following line declares a variable named `array` and creates an array of three characters:

```
char[] array = {'c', 'a', 't'};
```

Figure 9-2 shows them both, with a box to represent the location of the variable and an arrow pointing to the location of the array.

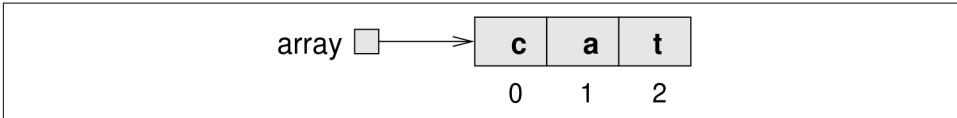


Figure 9-2. Memory diagram of an array of characters

Objects work in a similar way. For example, this line declares a `String` variable named `word` and creates a `String` object, as shown in Figure 9-3:

```
String word = "dog";
```

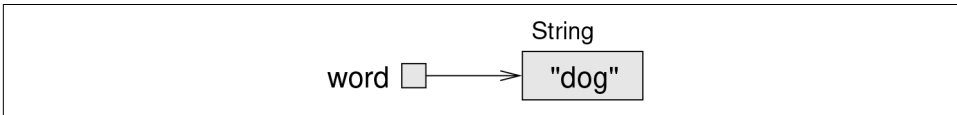


Figure 9-3. Memory diagram of a `String` object

Objects and arrays are usually created with the `new` keyword, which allocates memory for them. For convenience, you don’t have to use `new` to create strings:

```
String word = new String("dog"); // creates a string object  
String word = "dog"; // implicitly creates a string object
```

Recall from “String Comparison” on page 88 that you need to use the `equals` method to compare strings. The `equals` method traverses the `String` objects and tests whether they contain the same characters.

To test whether two integers or other primitive types are equal, you can simply use the `==` operator. But two `String` objects with the same characters would not be considered equal in the `==` sense. The `==` operator, when applied to string variables, tests only whether they refer to the *same* object.

The null Keyword

Often when you declare an object variable, you assign it to reference an object. But sometimes you want to declare a variable that doesn’t refer to an object, at least initially.

In Java, the keyword `null` is a special value that means *no object*. You can initialize object and array variables this way:

```
String name = null;
int[] combo = null;
```

The value `null` is represented in memory diagrams by a small box with no arrow, as in [Figure 9-4](#).

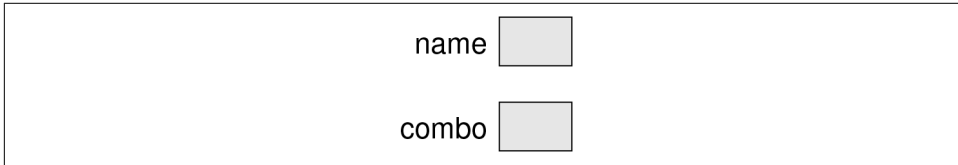


Figure 9-4. Memory diagram showing variables that are `null`

If you try to use a variable that is `null` by invoking a method or accessing an element, Java throws a `NullPointerException`:

```
System.out.println(name.length()); // NullPointerException
System.out.println(combo[0]);      // NullPointerException
```

On the other hand, it is perfectly fine to pass a `null` reference as an argument to a method, or to receive one as a return value. In these situations, `null` is often used to represent a special condition or indicate an error.

Strings Are Immutable

If the Java library didn't have a `String` class, we would have to use character arrays to store and manipulate text. Operations like concatenation (+), `indexOf`, and `substring` would be difficult and inconvenient. Fortunately, Java does have a `String` class that provides these and other methods.

For example, the methods `toLowerCase` and `toUpperCase` convert uppercase letters to lowercase, and vice versa. These methods are often a source of confusion, because it sounds like they modify strings. But neither these methods nor any others can change a string, because strings are **immutable**.

When you invoke `toUpperCase` on a string, you get a new `String` object as a result. For example:

```
String name = "Alan Turing";
String upperName = name.toUpperCase();
```

After these statements run, `upperName` refers to the string "ALAN TURING". But `name` still refers to "Alan Turing". A common mistake is to assume that `toUpperCase` somehow affects the original string:

```
String name = "Alan Turing";
name.toUpperCase(); // ignores the return value
System.out.println(name);
```

The previous code displays "Alan Turing", because the value of `name`, which refers to the original `String` object, never changes. If you want to change `name` to be uppercase, then you need to assign the return value:

```
String name = "Alan Turing";
name = name.toUpperCase(); // references the new string
System.out.println(name);
```

A similar method is `replace`, which finds and replaces instances of one string within another. This example replaces "Computer Science" with "CS":

```
String text = "Computer Science is fun!";
text = text.replace("Computer Science", "CS");
```

As with `toUpperCase`, assigning the return value (to `text`) is important. If you don't assign the return value, invoking `text.replace` has no effect.

Strings are immutable by design, because it simplifies passing them as parameters and return values. And since the contents of a string can never change, two variables can reference the same string without one accidentally corrupting the other.

Wrapper Classes

Primitive types like `int`, `double`, and `char` cannot be null, and they do not provide methods. For example, you can't invoke `equals` on an `int`:

```
int i = 5;
System.out.println(i.equals(5)); // compiler error
```

But for each primitive type, there is a corresponding **wrapper class** in the Java library. The wrapper class for `int` is named `Integer`, with a capital I:

```
Integer i = Integer.valueOf(5);
System.out.println(i.equals(5)); // displays true
```

Other wrapper classes include `Boolean`, `Character`, `Double`, and `Long`. They are in the `java.lang` package, so you can use them without importing them.

Like strings, objects from wrapper classes are immutable, and you have to use the `equals` method to compare them:

```

Integer x = Integer.valueOf(123);
Integer y = Integer.valueOf(123);
if (x == y) { // false
    System.out.println("x and y are the same object");
}
if (x.equals(y)) { // true
    System.out.println("x and y have the same value");
}

```

Because `x` and `y` refer to different objects, this code displays only “`x` and `y` have the same value”.

Each wrapper class defines the constants `MIN_VALUE` and `MAX_VALUE`. For example, `Integer.MIN_VALUE` is `-2147483648`, and `Integer.MAX_VALUE` is `2147483647`. Because these constants are available in wrapper classes, you don’t have to remember them, and you don’t have to write them yourself.

Wrapper classes also provide methods for converting strings to and from primitive types. For example, `Integer.parseInt` converts a string to an `int`. In this context, **parse** means *read and translate*.

```

String str = "12345";
int num = Integer.parseInt(str);

```

Other wrapper classes provide similar methods, like `Double.parseDouble` and `Boolean.parseBoolean`. They also provide `toString`, which returns a string representation of a value:

```

int num = 12345;
String str = Integer.toString(num);

```

The result is the `String` object `"12345"`.

It’s always possible to convert a primitive value to a string, but not the other way around. For example, say we try to parse an invalid string like this:

```

String str = "five";
int num = Integer.parseInt(str); // NumberFormatException

```

`parseInt` throws a `NumberFormatException`, because the characters in the string `"five"` are not digits.

Command-Line Arguments

Now that you know about strings, arrays, and wrapper classes, we can *finally* explain the `args` parameter of the `main` method, which we have been ignoring since [Chapter 1](#). If you are unfamiliar with the command-line interface, please read “[Command-Line Interface](#)” on page 261.

Let’s write a program to find the maximum value in a sequence of numbers. Rather than read the numbers from `System.in` by using a `Scanner`, we’ll pass them as command-line arguments. Here is a starting point:

```
import java.util.Arrays;
public class Max {
    public static void main(String[] args) {
        System.out.println(Arrays.toString(args));
    }
}
```

You can run this program from the command line by typing this:

```
java Max
```

The output indicates that `args` is an **empty array**; that is, it has no elements:

```
[]
```

If you provide additional values on the command line, they are passed as arguments to `main`. For example, say you run the program like this:

```
java Max 10 -3 55 0 14
```

The output is shown here:

```
[10, -3, 55, 0, 14]
```

It’s not clear from the output, but the elements of `args` are strings. So `args` is the array `{"10", "-3", "55", "0", "14"}`. To find the maximum number, we have to convert the arguments to integers.

The following code uses an enhanced for loop (see “[The Enhanced for Loop](#)” on page 104) to parse the arguments and find the largest value:

```
int max = Integer.MIN_VALUE;
for (String arg : args) {
    int value = Integer.parseInt(arg);
    if (value > max) {
        max = value;
    }
}
System.out.println("The max is " + max);
```


We begin by initializing `max` to the smallest (most negative) number an `int` can represent. That way, the first value we parse will replace `max`. As we find larger values, they will replace `max` as well.

If `args` is empty, the result will be `MIN_VALUE`. We can prevent this situation from happening by checking `args` at the beginning of the program:

```
if (args.length == 0) {
    System.err.println("Usage: java Max <numbers>");
    return;
}
```

It's customary for programs that require command-line arguments to display a *usage* message if the arguments are not valid. For example, if you run `javac` or `java` from the command line without any arguments, you will get a very long message.

Argument Validation

As we discussed in “Validating Input” on page 71, you should never assume that program input will be in the correct format. Sometimes users make mistakes, such as pressing the wrong key or misreading instructions.

Or even worse, someone might make intentional “mistakes” to see what your program will do. One way hackers break into computer systems is by entering malicious input that causes a program to fail.

Programmers can make mistakes too. It's difficult to write bug-free software, especially when working in teams on large projects.

For all of these reasons, it's good practice to validate arguments passed to methods, including the `main` method. In the previous section, we did this by ensuring that `args.length` was not 0.

As a further example, consider a method that checks whether the first word of a sentence is capitalized. We can write this method using the `Character` wrapper class:

```
public static boolean isCapitalized(String str) {
    return Character.isUpperCase(str.charAt(0));
}
```

The expression `str.charAt(0)` makes two assumptions: the string object referenced by `str` exists, and it has at least one character. What if these assumptions don't hold at run-time?

- If `str` is `null`, invoking `charAt` will cause a `NullPointerException`, because you can't invoke a method on `null`.

- If `str` refers to an empty string, which is a `String` object with no characters, `charAt` will cause a `StringIndexOutOfBoundsException`, because there is no character at index 0.

We can prevent these exceptions by validating `str` *at the start* of the method. If it's invalid, we return before executing the rest of the method:

```
public static boolean isCapitalized(String str) {
    if (str == null || str.isEmpty()) {
        return false;
    }
    return Character.isUpperCase(str.charAt(0));
}
```

Notice that `null` and *empty* are different concepts, as shown in [Figure 9-5](#). The variable `str1` is `null`, meaning that it doesn't reference an object. The variable `str2` refers to the empty string, an object that exists.

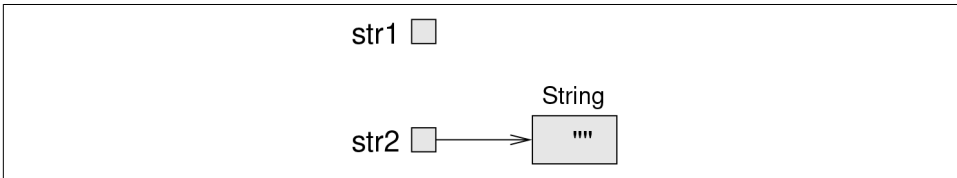


Figure 9-5. Memory diagram of `null` and empty string

Beginners sometimes make the mistake of checking for empty first. Doing so causes a `NullPointerException`, because you can't invoke methods on variables that are `null`:

```
if (str.isEmpty() || str == null) { // wrong!
```

Checking for `null` first prevents the `NullPointerException`. If `str` is `null`, the `||` operator will short-circuit (see [“Logical Operators” on page 67](#)) and evaluate to `true` immediately. As a result, `str.isEmpty()` will not be called.

BigInteger Arithmetic

It might not be clear at this point why you would ever need an integer object when you can just use an `int` or `long`. One advantage is the variety of methods that `Integer` and `Long` provide. But there is another reason: when you need very large integers that exceed `Long.MAX_VALUE`.

`BigInteger` is a Java class that can represent arbitrarily large integers. There is no upper bound except the limitations of memory size and processing speed. Take a minute to read the documentation, which you can find by doing a web search for “Java `BigInteger`”.

To use `BigInteger`s, you have to `import java.math.BigInteger` at the beginning of your program. There are several ways to create a `BigInteger`, but the simplest uses `valueOf`. The following code converts a `long` to a `BigInteger`:

```
long x = 17;
BigInteger big = BigInteger.valueOf(x);
```

You can also create `BigInteger`s from strings. For example, here is a 20-digit integer that is too big to store using a `long`:

```
String s = "12345678901234567890";
BigInteger bigger = new BigInteger(s);
```

Notice the difference in the previous two examples: you use `valueOf` to convert integers, and `new BigInteger` to convert strings.

Since `BigInteger`s are not primitive types, the usual math operators don't work. Instead, we have to use methods like `add`. To add two `BigInteger`s, we invoke `add` on one and pass the other as an argument:

```
BigInteger a = BigInteger.valueOf(17);
BigInteger b = BigInteger.valueOf(1700000000);
BigInteger c = a.add(b);
```

Like strings, `BigInteger` objects are immutable. Methods like `add`, `multiply`, and `pow` all return new `BigInteger`s, rather than modify an existing one.

Internally, a `BigInteger` is implemented using an array of `ints`, similar to the way a string is implemented using an array of `chars`. Each `int` in the array stores a portion of the `BigInteger`. The methods of `BigInteger` traverse this array to perform addition, multiplication, etc.

For very long floating-point values, take a look at `java.math.BigDecimal`. Interestingly, `BigDecimal` objects represent floating-point numbers internally by using a `BigInteger`!

Incremental Design

One challenge of programming, especially for beginners, is figuring out how to divide a program into methods. In this section, we present a **design process** that allows you to divide a program into methods as you go along. The process is called *encapsulation and generalization*. The essential steps are as follows:

1. Write a few lines of code in `main` or another method, and test them.
2. When they are working, wrap them in a new method and test again.
3. If it's appropriate, replace literal values with variables and parameters.

To demonstrate this process, we'll develop methods that display multiplication tables. We begin by writing and testing a few lines of code. Here is a loop that displays the multiples of two, all on one line:

```
for (int i = 1; i <= 6; i++) {
    System.out.printf("%4d", 2 * i);
}
System.out.println();
```

Each time through the loop, we display the value of $2 * i$, padded with spaces so it's four characters wide. Since we use `System.out.printf`, the output appears on a single line.

After the loop, we call `println` to print a newline character. Remember that in some environments, none of the output is displayed until the line is complete. The output of the code so far is shown here:

```
2  4  6  8 10 12
```

The next step is to **encapsulate** the code; that is, we *wrap* the code in a method:

```
public static void printRow() {
    for (int i = 1; i <= 6; i++) {
        System.out.printf("%4d", 2 * i);
    }
    System.out.println();
}
```

Finally, we **generalize** the method to print multiples of other numbers by replacing the constant value 2 with a parameter `n`. This step is called *generalization*, because it makes the method more general (less specific):

```
public static void printRow(int n) {
    for (int i = 1; i <= 6; i++) {
        System.out.printf("%4d", n * i); // generalized n
    }
    System.out.println();
}
```

Invoking this method with the argument 2 yields the same output as before. With the argument 3, the output is as follows:

```
3  6  9 12 15 18
```

By now, you can probably guess how we are going to display a multiplication table: we'll invoke `printRow` repeatedly with different arguments. In fact, we'll use another loop to iterate through the rows:

```
for (int i = 1; i <= 6; i++) {
    printRow(i);
}
```

And the output looks like this:

```
1  2  3  4  5  6
2  4  6  8 10 12
3  6  9 12 15 18
4  8 12 16 20 24
5 10 15 20 25 30
6 12 18 24 30 36
```

More Generalization

The previous result is similar to the *nested loops* approach in “[Nested Loops](#)” on page 83. However, the inner loop is now encapsulated in the `printRow` method. We can encapsulate the outer loop in a method too:

```
public static void printTable() {
    for (int i = 1; i <= 6; i++) {
        printRow(i);
    }
}
```

The initial version of `printTable` always displays six rows. We can generalize it by replacing the literal 6 with a parameter:

```
public static void printTable(int rows) {
    for (int i = 1; i <= rows; i++) { // generalized rows
        printRow(i);
    }
}
```

Here is the output of `printTable(7)`:

```
1  2  3  4  5  6
2  4  6  8 10 12
3  6  9 12 15 18
4  8 12 16 20 24
5 10 15 20 25 30
6 12 18 24 30 36
7 14 21 28 35 42
```

That’s better, but it always displays the same number of columns. We can generalize more by adding a parameter to `printRow`:

```
public static void printRow(int n, int cols) {
    for (int i = 1; i <= cols; i++) { // generalized cols
        System.out.printf("%4d", n * i);
    }
    System.out.println();
}
```

Now `printRow` takes two parameters: `n` is the value whose multiples should be displayed, and `cols` is the number of columns. Since we added a parameter to `printRow`, we also have to change the line in `printTable` where it is invoked:

```
public static void printTable(int rows) {
    for (int i = 1; i <= rows; i++) {
        printRow(i, rows);
    }
}
```

When this line executes, it evaluates `rows` and passes the value, which is 7 in this example, as an argument. In `printRow`, this value is assigned to `cols`. As a result, the number of columns equals the number of rows, so we get a square 7×7 table, instead of the previous 7×6 table.

When you generalize a method appropriately, you often find that it has capabilities you did not plan. For example, you might notice that the multiplication table is symmetric. Since $ab = ba$, all the entries in the table appear twice. You could save ink by printing half of the table, and you would have to change only *one line* of `printTable`:

```
printRow(i, i); // using i for both n and cols
```

This means the length of each row is the same as its row number. The result is a triangular multiplication table:

```
1
2  4
3  6  9
4  8 12 16
5 10 15 20 25
6 12 18 24 30 36
7 14 21 28 35 42 49
```

Generalization makes code more versatile, more likely to be reused, and sometimes easier to write.

Vocabulary

object-oriented

A way of organizing code and data into objects, rather than independent methods.

object

A collection of related data that comes with a set of methods that operate on the data.

primitive

A data type that stores a single value and provides no methods.

immutable

An object that, once created, cannot be modified. Strings are immutable by design.

wrapper class

Classes in `java.lang` that provide constants and methods for working with primitive types.

parse

In [Chapter 2](#), we defined *parse* as what the compiler does to analyze a program. Now you know that it means to read a string and interpret or translate it.

empty array

An array with no elements and a length of zero.

design process

A process for determining what methods a class or program should have.

encapsulate

To wrap data inside an object, or to wrap statements inside a method.

generalize

To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter).

Exercises

The code for this chapter is in the `ch09` directory of `ThinkJavaCode2`. See [“Using the Code Examples” on page xii](#) for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

Exercise 9-1.

The point of this exercise is to explore Java types and fill in some of the details that aren't covered in the chapter.

1. Create a new program named `Test.java` and write a `main` method that contains expressions that combine various types using the `+` operator. For example, what happens when you “add” a `String` and a `char`? Does it perform character addition or string concatenation? What is the type of the result?
2. Make a bigger copy of the following table and fill it in. At the intersection of each pair of types, you should indicate whether it is legal to use the `+` operator with these types, the operation that is performed (addition or concatenation), and the type of the result.

	boolean	char	int	double	String
boolean					
char					
int					
double					
String					

3. Think about some of the choices the designers of Java made, based on this table. How many of the entries seem unavoidable, as if there was no other choice? How many seem like arbitrary choices from several equally reasonable possibilities? Which entries seem most problematic?
4. Here's a puzzler: normally, the statement `x++` is exactly equivalent to `x = x + 1`. But if `x` is a `char`, it's not exactly the same! In that case, `x++` is legal, but `x = x + 1` causes an error. Try it. See what the error message is, and then see if you can figure out what is going on.
5. What happens when you add `""` (the empty string) to the other types; for example, `"" + 5`?

Exercise 9-2.

You might be sick of the `factorial` method by now, but we're going to do one more version.

1. Create a new program called `Big.java` and write an iterative version of `factorial` (using a `for` loop).
2. Display a table of the integers from 0 to 30 along with their factorials. At some point around 15, you will probably see that the answers are not correct anymore. Why not?
3. Convert `factorial` so that it performs its calculation using `BigInteger`s and returns a `BigInteger` as a result. You can leave the parameter alone; it will still be an integer.
4. Try displaying the table again with your modified `factorial` method. Is it correct up to 30? How high can you make it go?

Exercise 9-3.

Many encryption algorithms depend on the ability to raise large integers to a power. Here is a method that implements an efficient algorithm for integer exponentiation:


```

public static int pow(int x, int n) {
    if (n == 0) return 1;

    // find x to the n/2 recursively
    int t = pow(x, n / 2);

    // if n is even, the result is t squared
    // if n is odd, the result is t squared times x
    if (n % 2 == 0) {
        return t * t;
    } else {
        return t * t * x;
    }
}

```

The problem with this method is that it works only if the result is small enough to be represented by an `int`. Rewrite it so that the result is a `BigInteger`. The parameters should still be integers, though.

You should use the `BigInteger` methods `add` and `multiply`. But don't use `BigInteger.pow`; that would spoil the fun.

Exercise 9-4.

One way to calculate e^x is to use the following infinite series expansion. The i th term in the series is $x^i/i!$.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

1. Write a method called `myexp` that takes `x` and `n` as parameters and estimates e^x by adding the first `n` terms of this series. You can use the `factorial` method from “Value-Returning Methods” on page 114 or your iterative version from the previous exercise.
2. You can make this method more efficient by observing that the numerator of each term is the same as its predecessor multiplied by `x`, and the denominator is the same as its predecessor multiplied by `i`.

Use this observation to eliminate the use of `Math.pow` and `factorial`, and check that you get the same result.

3. Write a method called `check` that takes a parameter, `x`, and displays `x`, `myexp(x)`, and `Math.exp(x)`. The output should look something like this:

```

1.0      2.708333333333333      2.718281828459045

```

You can use the escape sequence `"\t"` to put a tab character between columns of a table.

4. Vary the number of terms in the series (the second argument that `check` sends to `myexp`) and see the effect on the accuracy of the result. Adjust this value until the estimated value agrees with the correct answer when x is 1.
5. Write a loop in `main` that invokes `check` with the values `0.1`, `1.0`, `10.0`, and `100.0`. How does the accuracy of the result vary as x varies? Compare the number of digits of agreement rather than the difference between the actual and estimated values.
6. Add a loop in `main` that checks `myexp` with the values `-0.1`, `-1.0`, `-10.0`, and `-100.0`. Comment on the accuracy.

Exercise 9-5.

The goal of this exercise is to practice encapsulation and generalization using some of the examples in previous chapters.

1. Starting with the code in [“Traversing Arrays” on page 100](#), write a method called `powArray` that takes a `double` array, `a`, and returns a new array that contains the elements of `a` squared. Generalize it to take a second argument and raise the elements of `a` to the given power.
2. Starting with the code in [“The Enhanced for Loop” on page 104](#), write a method called `histogram` that takes an `int` array of scores from 0 to (but not including) 100, and returns a histogram of 100 counters. Generalize it to take the number of counters as an argument.

Exercise 9-6.

The following code fragment traverses a string and checks whether it has the same number of opening and closing parentheses:

```
String s = "((3 + 7) * 2)";
int count = 0;

for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    if (c == '(') {
        count++;
    } else if (c == ')') {
        count--;
    }
}

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

1. Encapsulate this fragment in a method that takes a string argument and returns the final value of count.
2. Test your method with multiple strings, including some that are balanced and some that are not.
3. Generalize the code so that it works on any string. What could you do to generalize it more?

Mutable Objects

As you learned in the previous chapter, an *object* is a collection of data that provides a set of methods. For example, a `String` is a collection of characters that provides methods like `charAt` and `substring`.

This chapter explores two new types of objects: `Point` and `Rectangle`. You'll see how to write methods that take objects as parameters and produce objects as return values. You will also take a first look at the source code for the Java library.

Point Objects

In math, 2D *points* are often written in parentheses with a comma separating the coordinates. For example, $(0, 0)$ indicates the origin, and (x, y) indicates the point x units to the right and y units up from the origin.

The `java.awt` package provides a class named `Point` that represents a location in a Cartesian plane. In order to use the `Point` class, you have to import it:

```
import java.awt.Point;
```

Then, to create a new point, you use the `new` operator:

```
Point blank;  
blank = new Point(3, 4);
```

The first line declares that `blank` has type `Point`. The second line creates the new `Point` with the coordinates $x = 3$ and $y = 4$. The result of the `new` operator is a *reference* to the object. [Figure 10-1](#) shows the result.

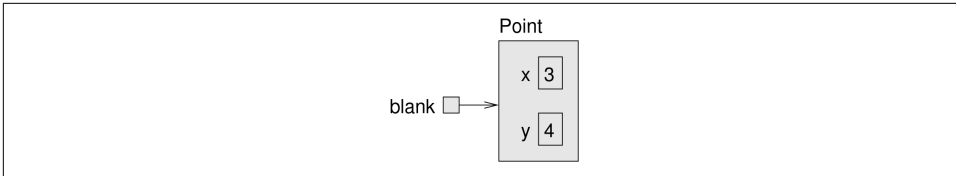


Figure 10-1. Memory diagram showing a variable that refers to a `Point` object

As usual, the name of the variable `blank` appears outside the box, and its value appears inside the box. In this case, the value is a reference, which is represented with an arrow. The arrow points to the `Point` object, which contains two variables, `x` and `y`.

Variables that belong to an object are called **attributes**. In some documentation, you also see them called *fields*. To access an attribute of an object, Java uses **dot notation**. For example:

```
int x = blank.x;
```

The expression `blank.x` means “go to the object `blank` refers to, and get the value of the attribute `x`.” In this case, we assign that value to a local variable named `x`.

There is no conflict between the local variable `x` and the attribute `x`. The purpose of dot notation is to identify *which* variable you are referring to unambiguously.

You can use dot notation as part of an expression. For example:

```
System.out.println(blank.x + ", " + blank.y);
int sum = blank.x * blank.x + blank.y * blank.y;
```

The first line displays `3, 4`. The second line calculates the value `25`.

Objects as Parameters

You can pass objects as parameters in the usual way. For example:

```
public static void printPoint(Point p) {
    System.out.println("(" + p.x + ", " + p.y + ")");
}
```

This method takes a point as an argument and displays its attributes in parentheses. If you invoke `printPoint(blank)`, it displays `(3, 4)`.

As another example, we can rewrite the `distance` method from “[Incremental Development](#)” on page 54 so that it takes two `Points` as parameters instead of four doubles:

```
public static double distance(Point p1, Point p2) {
    int dx = p2.x - p1.x;
    int dy = p2.y - p1.y;
    return Math.sqrt(dx * dx + dy * dy);
}
```

Passing objects as parameters makes the source code more readable and less error-prone because related values are bundled together.

You actually don't need to write a distance method, because `Point` objects already have one. To compute the distance between two points, we invoke `distance` on one and pass the other as an argument:

```
Point p1 = new Point(0, 0);
Point p2 = new Point(3, 4);
double dist = p1.distance(p2); // dist is 5.0
```

It turns out you don't need the `printPoint` method either. If you invoke `System.out.println(blank)`, it prints the type of the object and the values of the attributes:

```
java.awt.Point[x=3,y=4]
```

`Point` objects provide a method called `toString` that returns a string representation of a point. When you call `println` with objects, it *automatically* calls `toString` and displays the result.

Objects as Return Values

The `java.awt` package also provides a class named `Rectangle`. To use it, you have to import it:

```
import java.awt.Rectangle;
```

`Rectangle` objects are similar to points, but they have four attributes: `x`, `y`, `width`, and `height`. The following example creates a `Rectangle` object and makes the variable `box` refer to it:

```
Rectangle box = new Rectangle(0, 0, 100, 200);
```

Figure 10-2 shows the effect of this assignment.

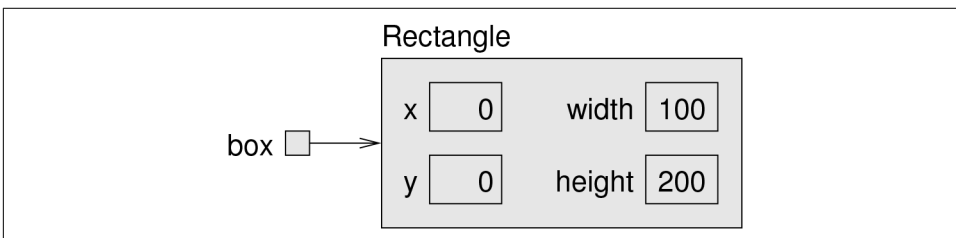


Figure 10-2. Memory diagram showing a `Rectangle` object

If you run `System.out.println(box)`, you get this:

```
java.awt.Rectangle[x=0,y=0,width=100,height=200]
```

Again, `println` uses the `toString` method provided by `Rectangle`, which knows how to represent `Rectangle` objects as strings.

You can also write methods that return new objects. For example, `findCenter` takes a `Rectangle` as an argument and returns a `Point` with the coordinates of the center of the rectangle:

```
public static Point findCenter(Rectangle box) {  
    int x = box.x + box.width / 2;  
    int y = box.y + box.height / 2;  
    return new Point(x, y);  
}
```

The return type of this method is `Point`. The last line creates a new `Point` object and returns a reference to it.

Rectangles Are Mutable

You can change the contents of an object by making an assignment to one of its attributes. For example, to *move* a rectangle without changing its size, you can modify the `x` and `y` values:

```
Rectangle box = new Rectangle(0, 0, 100, 200);  
box.x = box.x + 50;  
box.y = box.y + 100;
```

The result is shown in [Figure 10-3](#).

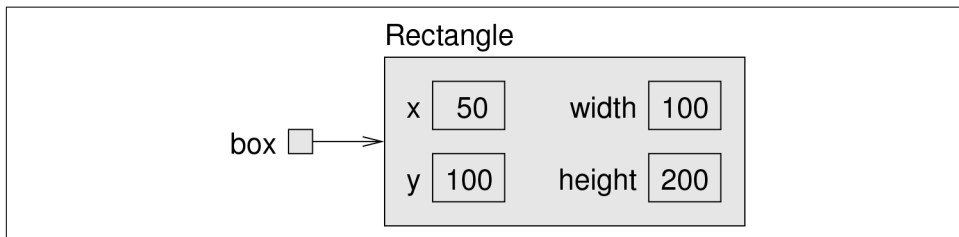


Figure 10-3. Memory diagram showing updated attributes

We can encapsulate this code in a method and generalize it to move the rectangle by any amount:

```
public static void moveRect(Rectangle box, int dx, int dy) {  
    box.x = box.x + dx;  
    box.y = box.y + dy;  
}
```


The variables `dx` and `dy` indicate how far to move the rectangle in each direction. Invoking this method has the effect of modifying the `Rectangle` that is passed as an argument:

```
Rectangle box = new Rectangle(0, 0, 100, 200);
moveRect(box, 50, 100); // now at (50, 100, 100, 200)
```

Modifying objects by passing them as arguments to methods can be useful. But it can also make debugging difficult, because it is not always clear which method invocations modify their arguments.

Java provides a number of methods that operate on `Points` and `Rectangles`. For example, `translate` has the same effect as `moveRect`, but instead of passing the rectangle as an argument, you use dot notation:

```
box.translate(50, 100);
```

This line invokes the `translate` method on the object that `box` refers to, which modifies the object.

This syntax—using dot notation to invoke a method on an object, rather than passing it as a parameter—is more consistent with the style of object-oriented programming.

Aliasing Revisited

Remember that when you assign an object to a variable, you are assigning a *reference* to an object. It is possible to have multiple variables that refer to the same object. For example, this code creates two variables that refer to the same `Rectangle`:

```
Rectangle box1 = new Rectangle(0, 0, 100, 200);
Rectangle box2 = box1;
```

Figure 10-4 shows the result: `box1` and `box2` refer to the same object, so any changes that affect one variable also affect the other.

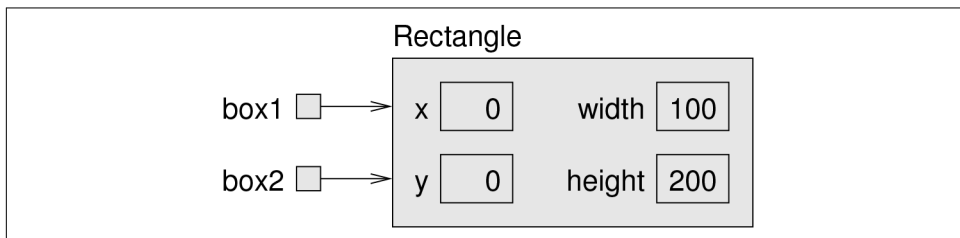


Figure 10-4. Memory diagram of two variables that refer to the same `Rectangle` object

For example, the following code uses `grow` to make `box1` bigger by 50 units in all directions. It decreases `x` and `y` by 50, and it increases `height` and `width` by 100:

```
box1.grow(50, 50); // grow box1 (alias)
```

The result is shown in [Figure 10-5](#).

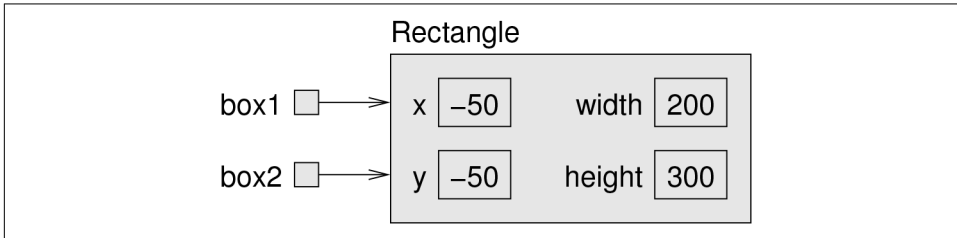


Figure 10-5. Memory diagram showing the effect of invoking `grow`

Now, if we print `box1`, we are not surprised to see that it has changed:

```
java.awt.Rectangle[x=-50,y=-50,width=200,height=300]
```

And if we print `box2`, we should not be surprised to see that it has changed too, because it refers to the same object:

```
java.awt.Rectangle[x=-50,y=-50,width=200,height=300]
```

This scenario is called *aliasing* because a single object has multiple names, or aliases, that refer to it.

As you can tell from this simple example, code that involves aliasing can get confusing fast, and it can be difficult to debug.

Java Library Source

So far we have used several classes from the Java library, including `System`, `String`, `Scanner`, `Math`, and `Random`. These classes are written in Java, so you can read the source code to see how they work.

The Java library contains thousands of files, many of which are thousands of lines of code. That's more than one person could read and understand fully, but don't be intimidated!

Because it's so large, the library source code is stored in a ZIP archive named `src.zip`. If you have Java installed on your computer, you should already have this file somewhere:

- On Linux, it's likely under `/usr/lib/jvm/.../lib`. If not, you might have to install the `openjdk-...-source` package.
- On macOS, it's likely under `/Library/Java/JavaVirtualMachines/.../Contents/Home/lib`.
- On Windows, it's likely under `C:\Program Files\Java\...\lib`.

When you open (or unzip) the file, you will see folders that correspond to Java packages. For example, open the *java* folder, and then open the *awt* folder. (If you don't see a *java* folder at first, open the *java.desktop* folder.) You should now see *Point.java* and *Rectangle.java*, along with the other classes in the *java.awt* package.

Open *Point.java* in your editor and skim through the file. It uses language features we haven't discussed yet, so you probably won't understand every line. But you can get a sense of what professional Java source code looks like by browsing through the library.

Notice how much of *Point.java* is documentation (see [Appendix B](#)). Each method includes comments and tags like `@param` and `@return`. Javadoc reads these comments and generates documentation in HTML. You can see the same documentation online by doing a web search for *Java Point*.

Now take a look at the `grow` and `translate` methods in the `Rectangle` class. There is more to them than you may have expected.

Class Diagrams

To summarize what you've learned so far, `Point` and `Rectangle` objects have attributes and methods. Attributes are an object's *data*; methods are an object's *code*. An object's *class* definition specifies the attributes and methods that it has.

Unified Modeling Language (UML) defines a graphical way to summarize this information. [Figure 10-6](#) shows two examples, the UML **class diagrams** for the `Point` and `Rectangle` classes.

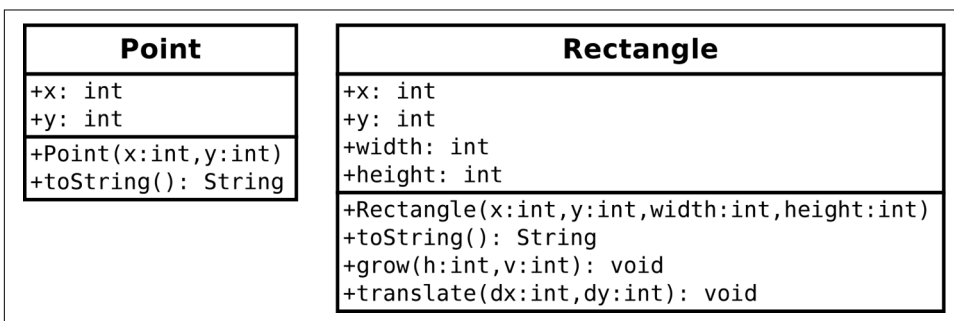


Figure 10-6. UML class diagrams for `Point` and `Rectangle`

Each class is represented by a box with the name of the class, a list of attributes, and a list of methods.

To identify the types of attributes and parameters, UML uses a language-independent syntax, like `x: int` rather than Java syntax, `int x`.

The plus sign (+) identifies public attributes and methods. A minus sign (-) identifies private attributes and methods, which we discuss in the next chapter.

Both `Point` and `Rectangle` have additional methods; we show only the ones introduced in this chapter.

In contrast to memory diagrams, which visualize objects (and variables) at run-time, a class diagram visualizes the source code at compile-time.

Scope Revisited

In “[Stack Diagrams](#)” on page 50, we introduced the idea that variables have scope. The scope of a variable is the part of a program where a variable can be used.

Consider the first few lines of the `Rectangle.translate` method from the Java library source code:

```
public void translate(int dx, int dy) {
    int oldv = this.x;
    int newv = oldv + dx;
    if (dx < 0) {
        ...
    }
}
```

This example uses three kinds of variables:

- Parameters (`dx` and `dy`)
- Local variables (`oldv` and `newv`)
- Attributes (`this.x`)

Parameters and local variables are created when a method is invoked, and they disappear when the method returns. They can be used anywhere inside the method, but not in other methods and not in other classes.

Attributes are created when an object is created, and they disappear when the object is destroyed. They can be used in any of the object’s methods, using the keyword `this`. And if they are public, they can be used in other classes via references to the object, `box1.x`.

When the Java compiler encounters a variable name, it searches backward for its declaration. The compiler first looks for local variables, then parameters, then attributes.

Garbage Collection

In the previous section, we said that attributes exist as long as the object exists. But when does an object cease to exist? Here is a simple example:

```
Point blank = new Point(3, 4);
blank = null;
```

The first line creates a new `Point` object and makes `blank` refer to it. The second line changes `blank` so that instead of referring to the object, it refers to nothing. As shown in [Figure 10-7](#), after the second assignment, there are no references to the `Point` object.

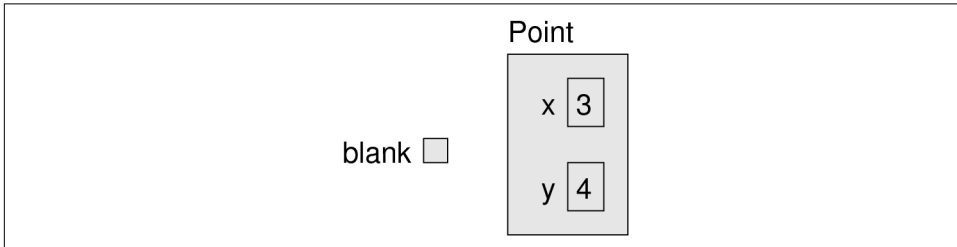


Figure 10-7. Memory diagram showing the effect of setting a variable to `null`

If there are no references to an object, there is no way to access its attributes or invoke a method on it. From the program's point of view, it ceases to exist. However, it's still present in the computer's memory, taking up space.

As your program runs, the system automatically looks for stranded objects and deletes them; then the space can be reused for new objects. This process is called **garbage collection**.

You don't have to do anything to make garbage collection happen, and in general, you don't have to be aware of it. But in high-performance applications, you may notice a slight delay every now and then while Java reclaims space from discarded objects.

Mutable Versus Immutable

`Point` and `Rectangle` are **mutable** objects, because their attributes can be modified. You can modify their attributes directly, like `box.x = 15`, or you can invoke methods that modify their attributes, like `box.translate(15, 0)`.

In contrast, immutable objects like `String` and `Integer` cannot be modified. They don't allow direct access to their attributes or provide methods that change them.

Immutable objects have advantages that help improve the reliability and performance of programs. You can pass strings (and other immutable objects) to methods without worrying about their contents changing as a side-effect of the method. That makes programs easier to debug and more reliable.

Also, two strings that contain the same characters can be stored in memory only once. That can reduce the amount of memory the program uses and can speed it up.

In the following example, `s1` and `s2` are created differently, but they refer to equivalent strings; that is, the two strings contain the same characters:

```
public class Surprise {
    public static void main(String[] args) {
        String s1 = "Hi, Mom!";
        String s2 = "Hi, " + "Mom!";
        if (s1 == s2) { // true!
            System.out.println("s1 and s2 are the same");
        }
    }
}
```

Because both strings are specified at compile time, the compiler can tell that they are equivalent. And because strings are immutable, there is no need to make two copies; the compiler can create one `String` and make both variables refer to it.

As a result, the test `s1 == s2` turns out to be true, which means that `s1` and `s2` refer to the same object. In other words, they are not just equivalent; they are identical.

Although immutable objects have some advantages, mutable objects have other advantages. Sometimes it is more efficient to modify an existing object, rather than create a new one. And some computations can be expressed more naturally using mutation. Neither design is always better, which is why you will see both.

StringBuilder Objects

Here's an example in which mutable objects are efficient and arguably more natural: building a long string by concatenating lots of small pieces.

Strings are particularly inefficient for this operation. For example, consider the following program, which reads 10 lines from `System.in` and concatenates them into a single `String`:

```
String text = "";
for (int i = 0; i < 10; i++) {
    String line = in.nextLine(); // new string
    text = text + line + '\n'; // two more strings
}
```

Inside the `for` loop, `in.nextLine()` returns a new string each time it is invoked. The next line of code concatenates `text` and `line`, which creates another string, and then appends the newline character, which creates yet another string.

As a result, this loop creates 30 `String` objects! At the end, `text` refers to the most recent `String`. Garbage collection deletes the rest, but that's a lot of garbage for a seemingly simple program.

The Java library provides the `StringBuilder` class for just this reason. It's part of the `java.lang` package, so you don't need to import it. Because `StringBuilder` objects are mutable, they can implement concatenation more efficiently.

Here's a version of the program that uses `StringBuilder`:

```
StringBuilder text = new StringBuilder();
for (int i = 0; i < 10; i++) {
    String line = in.nextLine();
    text.append(line);
    text.append('\n');
}
```

The `append` method takes a `String` as a parameter and appends it to the end of the `StringBuilder`. Each time it is invoked, it modifies the `StringBuilder`; it doesn't create any new objects.

The `StringBuilder` class also provides methods for inserting and deleting parts of strings efficiently. Programs that manipulate large amounts of text run much faster if you use `StringBuilder` instead of `String`.

Vocabulary

attribute

One of the named data items that make up an object.

dot notation

Use of the dot operator (`.`) to access an object's attributes or methods.

UML

Unified Modeling Language, a standard way to draw diagrams for software engineering.

class diagram

An illustration of the attributes and methods for a class.

garbage collection

The process of finding objects that have no references and reclaiming their storage space.

mutable

An object that can be modified at any time. Points and rectangles are mutable by design.

Exercises

The code for this chapter is in the *ch10* directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

At this point, you know enough to read [Appendix C](#), which is about simple 2D graphics and animations. During the next few chapters, you should take a detour to read this appendix and work through the exercises.

Exercise 10-1.

The point of this exercise is to make sure you understand the mechanism for passing objects as parameters.

1. For the following program, draw a stack diagram showing the local variables and parameters of `main` and `riddle` just before `riddle` returns. Use arrows to show which objects each variable references.
2. What is the output of the program?
3. Is the `blank` object mutable or immutable? How can you tell?

```
public static int riddle(int x, Point p) {
    x = x + 7;
    return x + p.x + p.y;
}

public static void main(String[] args) {
    int x = 5;
    Point blank = new Point(1, 2);

    System.out.println(riddle(x, blank));
    System.out.println(x);
    System.out.println(blank.x);
    System.out.println(blank.y);
}
```

Exercise 10-2.

The point of this exercise is to make sure you understand the mechanism for returning new objects from methods. The following code uses `findCenter` and `distance` as defined in this chapter.

1. Draw a stack diagram showing the state of the program just before `findCenter` returns. Include all variables and parameters, and show the objects those variables refer to.

2. Draw a stack diagram showing the state of the program just before `distance` returns. Show all variables, parameters, and objects.
3. What is the output of this program? (Can you tell without running it?)

```
public static void main(String[] args) {
    Point blank = new Point(5, 8);

    Rectangle rect = new Rectangle(0, 2, 4, 4);
    Point center = findCenter(rect);

    double dist = distance(center, blank);
    System.out.println(dist);
}
```

Exercise 10-3.

This exercise is about aliasing. Recall that aliases are two variables that refer to the same object. The following code uses `findCenter` and `printPoint` as defined in this chapter.

1. Draw a diagram that shows the state of the program just before the end of `main`. Include all local variables and the objects they refer to.
2. What is the output of the program?
3. At the end of `main`, are `p1` and `p2` aliased? Why or why not?

```
public static void main(String[] args) {
    Rectangle box1 = new Rectangle(2, 4, 7, 9);
    Point p1 = findCenter(box1);
    printPoint(p1);

    box1.grow(1, 1);
    Point p2 = findCenter(box1);
    printPoint(p2);
}
```

Designing Classes

Whenever you create a new class, you are creating a new object type with the same name. So way back in “[The Hello World Program](#)” on page 3, when we created the class `Hello`, we also created an object type named `Hello`.

We didn’t declare any variables with type `Hello`, and we didn’t use `new` to create `Hello` objects. And it wouldn’t have done much good if we had—but we could have!

In this chapter, you will learn to design classes that represent *useful* objects. Here are the main ideas:

- Again, defining a **class** creates a new object type with the same name.
- A class definition is a template for objects: it specifies what attributes the objects have and what methods can operate on them.
- Every object belongs to an object type; that is, it is an **instance** of a class.
- The `new` operator **instantiates** objects; that is, it creates new instances of a class.

Think of a class as a blueprint for a house: you can use the same blueprint to build any number of houses.

The Time Class

A common reason to define a new class is to encapsulate related data in an object that can be treated as a single unit. That way, we can use objects as parameters and return values, rather than passing and returning multiple values. You have already seen two types that encapsulate data in this way: `Point` and `Rectangle`.

Another example, which we will implement ourselves, is `Time`, which represents a time of day. The data encapsulated in a `Time` object includes an hour, a minute, and a

number of seconds. Because every `Time` object contains these values, we define attributes to hold them.

Attributes are also called **instance variables**, because each instance has its own variables (as opposed to *class variables*, coming up in “Class Variables” on page 178).

The first step is to decide what type each variable should be. It seems clear that `hour` and `minute` should be integers. Just to keep things interesting, let’s make `second` a double.

Instance variables are declared at the beginning of the class definition, outside any method. By itself, this code fragment is a legal class definition:

```
public class Time {
    private int hour;
    private int minute;
    private double second;
}
```

The `Time` class is `public`, which means that it can be used in other classes. But the instance variables are `private`, which means they can be accessed only from inside the `Time` class. If you try to read or write them from another class, you will get a compiler error.

Private instance variables help keep classes isolated from each other, so that changes in one class won’t require changes in other classes. It also simplifies what other programmers need to know to use your classes. This kind of isolation is called **information hiding**.

Constructors

After declaring instance variables, the next step is to define a **constructor**, which is a special method that initializes the object. The syntax for constructors is similar to that of other methods, except for the following:

- The name of the constructor is the same as the name of the class.
- Constructors have no return type (and no return value).
- The keyword `static` is omitted.

Here is an example constructor for the `Time` class:

```
public Time() {
    this.hour = 0;
    this.minute = 0;
    this.second = 0.0;
}
```

This constructor does not take any arguments. Each line initializes an instance variable to 0 (which is midnight for a `Time` object).

The name `this` is a keyword that refers to the object we are creating. You can use `this` the same way you use the name of any other object. For example, you can read and write the instance variables of `this`, and you can pass `this` as an argument to other methods. But you do not declare `this`, and you can't make an assignment to it.

A common error when writing constructors is to put a `return` statement at the end. Like `void` methods, constructors do not return values.

To create a `Time` object, you must use the `new` operator:

```
public static void main(String[] args) {  
    Time time = new Time();  
}
```

When you use `new`, Java creates the object and invokes your constructor to initialize the instance variables. When the constructor is done, `new` returns a reference to the new object. In this example, the reference gets assigned to the variable `time`, which has type `Time`. [Figure 11-1](#) shows the result.

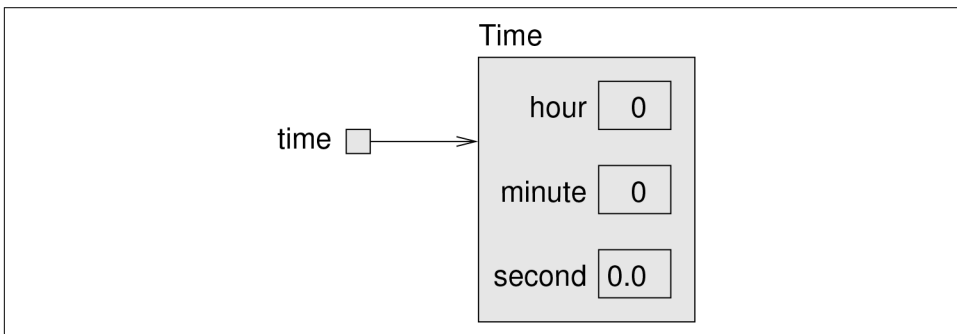


Figure 11-1. Memory diagram of a `Time` object

Beginners sometimes make the mistake of using `new` in the constructor:

```
public Time() {  
    new Time();           // StackOverflowError  
    this.hour = 0;  
    this.minute = 0;  
    this.second = 0.0;  
}
```

Doing so causes an infinite recursion, since `new` invokes the *same* constructor, which uses `new` again, which invokes the constructor again, and so on.

Value Constructors

Like other methods, constructors can be overloaded, which means you can provide multiple constructors with different parameters. Java knows which constructor to invoke by matching the arguments you provide with the parameters of the constructor.

It is common to provide both a *default constructor* that takes no arguments, like the previous one, and a *value constructor*, like this one:

```
public Time(int hour, int minute, double second) {  
    this.hour = hour;  
    this.minute = minute;  
    this.second = second;  
}
```

To invoke this constructor, you have to provide arguments to the new operator. The following example creates a `Time` object that represents a fraction of a second before noon:

```
Time time = new Time(11, 59, 59.9);
```

Overloading constructors provides the flexibility to create an object first and then fill in the attributes, or collect all the information before creating the object itself.

Once you get the hang of it, writing constructors gets boring. You can write them quickly just by looking at the list of instance variables. In fact, some IDEs can generate them for you.

Here is the complete class definition so far:

```
public class Time {  
    private int hour;  
    private int minute;  
    private double second;  
  
    public Time() {  
        this.hour = 0;  
        this.minute = 0;  
        this.second = 0.0;  
    }  
  
    public Time(int hour, int minute, double second) {  
        this.hour = hour;  
        this.minute = minute;  
        this.second = second;  
    }  
}
```

Notice how the second constructor declares the parameters `hour`, `minute`, and `sec` on. Java allows you to declare parameters (and local variables) with the same names as instance variables. They don't have to use the same names, but it's common practice.

The right side of `this.hour = hour;` refers to the parameter `hour`, since it was declared most recently. This situation is called **shadowing**, because the parameter “hides” the instance variable with the same name.

Java provides the keyword `this` so you can access instance variables, regardless of shadowing. As a result, this constructor copies the values from the parameters to the instance variables.

Getters and Setters

Recall that the instance variables of `Time` are `private`. We can access them from within the `Time` class, but if we try to read or write them from another class, the compiler reports an error.

A class that uses objects defined in another class is called a **client**. For example, here is a new class called `TimeClient`:

```
public class TimeClient {  
  
    public static void main(String[] args) {  
        Time time = new Time(11, 59, 59.9);  
        System.out.println(time.hour);    // compiler error  
    }  
}
```

If you compile this code, you get an error message like `hour has private access in Time`. There are three ways to solve this problem:

- Make the instance variables `public`.
- Provide methods to access the instance variables.
- Decide that it's not a problem and refuse to let other classes access the instance variables.

The first choice is appealing because it's simple. But here is the problem: when class *A* accesses the instance variables of class *B* directly, *A* becomes dependent on *B*. If anything in *B* changes later, it is likely that *A* will have to change too.

But if *A* uses only methods to interact with *B*, *A* and *B* are less dependent, which means that we can make changes in *B* without affecting *A* (as long as we don't change the method parameters). So we generally avoid making instance variables `public`.

The second option is to provide methods that access the instance variables. For example, we might want the instance variables to be *read only*; that is, code in other classes should be able to read them but not write them. We can do that by providing one method for each instance variable:

```
public int getHour() {
    return this.hour;
}

public int getMinute() {
    return this.minute;
}

public double getSecond() {
    return this.second;
}
```

Methods like these are formally called *accessors*, but more commonly referred to as **getters**. By convention, the method that gets a variable named something is called `getSomething`.

We can fix the compiler error in `TimeClient` by using the getter:

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

If we decide that `TimeClient` should also be able to modify the instance variables of `Time`, we can provide methods to do that too:

```
public void setHour(int hour) {
    this.hour = hour;
}

public void setMinute(int minute) {
    this.minute = minute;
}

public void setSecond(double second) {
    this.second = second;
}
```

These methods are formally called *mutators*, but more commonly known as **setters**. The naming convention is similar; the method that sets something is usually called `setSomething`.

Writing getters and setters can get boring, but many IDEs can generate them for you based on the instance variables.

Displaying Objects

To display `Time` objects, we can write a method to display the hour, minute, and second. Using `printTime` in “Multiple Parameters” on page 49 as a starting point, we could write the following:

```
public static void printTime(Time t) {
    System.out.print(t.hour);
    System.out.print(":");
    System.out.print(t.minute);
    System.out.print(":");
    System.out.println(t.second);
}
```

The output of this method, given the `time` object from the first example, would be `11:59:59.9`. We can use `printf` to make the code more concise:

```
public static void printTime(Time t) {
    System.out.printf("%02d:%02d:%04.1f\n",
        t.hour, t.minute, t.second);
}
```

As a reminder, you need to use `%d` with integers, and `%f` with floating-point numbers. The `02` option means “total width 2, with leading zeros if necessary”, and the `04.1` option means “total width 4, one digit after the decimal point, leading zeros if necessary”. The output is the same: `11:59:59.9`.

There’s nothing wrong with a method like `printTime`, but it is not consistent with object-oriented style. A more idiomatic solution is to provide a special method called `toString`.

The `toString` Method

Every object has a method called `toString` that returns a string representation of the object. When you display an object using `print` or `println`, Java invokes the object’s `toString` method.

By default, it simply displays the type of the object and its address in hexadecimal. So, say you create a `Time` object and display it with `println`:

```
public static void main(String[] args) {
    Time time = new Time(11, 59, 59.9);
    System.out.println(time);
}
```

The output looks something like this:

```
Time@80cc7c0
```

This address can be useful for debugging, if you want to keep track of individual objects.

But you can **override** this behavior by providing your own `toString` method. For example, here is a `toString` method for `Time`:

```
public String toString() {  
    return String.format("%02d:%02d:%04.1f\n",  
        this.hour, this.minute, this.second);  
}
```

The definition does not have the keyword `static`, because it is not a static method. It is an **instance method**, so called because when you invoke it, you invoke it on an instance of the class. Instance methods are sometimes called *nonstatic*; you might see this term in an error message.

The body of the method is similar to `printTime` in the previous section, with two changes:

- Inside the method, we use `this` to refer to the current instance; that is, the object the method is invoked on.
- Instead of `printf`, it uses `String.format`, which returns a formatted `String` rather than displaying it.

Now you can call `toString` directly:

```
Time time = new Time(11, 59, 59.9);  
String s = time.toString();
```

The value of `s` is the string `"11:59:59.9"`. You can also invoke `toString` indirectly by invoking `print` or `println`:

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

This code displays the string `"11:59:59.9"`. Either way, when you use `this` inside `toString`, it refers to the same object as `time`.

The equals Method

We have seen two ways to check whether values are equal: the `==` operator and the `equals` method. With objects, you can use either one, but they are not the same:

- The `==` operator checks whether two references are **identical**; that is, whether they refer to the same object.
- The `equals` method checks whether two objects are **equivalent**; that is, whether they have the same values.

The definition of *identity* is always the same, so the `==` operator always does the same thing. But the definition of *equivalence* is different for different objects, so objects can define their own `equals` methods.

Consider the following variables, depicted in the memory diagram in [Figure 11-2](#):

```
Time time1 = new Time(9, 30, 0.0);
Time time2 = time1;
Time time3 = new Time(9, 30, 0.0);
```

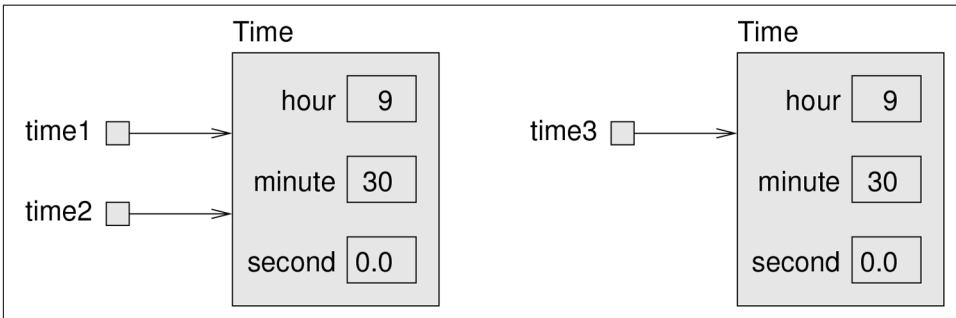


Figure 11-2. Memory diagram of three `Time` variables

The assignment operator copies references, so `time1` and `time2` refer to the same object. Because they are identical, `time1 == time2` is true. But `time1` and `time3` refer to two different objects. Because they are not identical, `time1 == time3` is false.

By default, the `equals` method does the same thing as `==`. For `Time` objects, that's probably not what we want. For example, `time1` and `time3` represent the same time of day, so we should consider them equivalent.

We can provide an `equals` method that implements this idea:

```
public boolean equals(Time that) {
    final DELTA = 0.001;
    return this.hour == that.hour
        && this.minute == that.minute
        && Math.abs(this.second - that.second) < DELTA;
}
```

`equals` is an instance method, so it doesn't have the keyword `static`. It uses `this` to refer to the current object, and `that` to refer to the other. `that` is *not* a keyword, so we could have given this parameter a different name. But using `that` makes the code nicely readable.

We can invoke `equals` like this:

```
time1.equals(time3);
```

Inside the `equals` method, `this` refers to the same object as `time1`, and `that` refers to the same object as `time3`. Since their instance variables are *equal*, the result is `true`.

Because `hour` and `minute` are integers, we compare them with `==`. But `second` is a floating-point number. Because of rounding errors, it is not good to compare floating-point numbers with `==` (see “[Rounding Errors](#)” on page 21). Instead, we check whether the difference is smaller than a threshold, `DELTA`.

Many objects have a similar notion of equivalence; that is, two objects are considered equal if their instance variables are equal. But other definitions are possible.

Adding Times

Suppose you are going to a movie that starts at 18:50 (i.e., 6:50 p.m.), and the running time is 2 hours, 16 minutes. What time does the movie end? We’ll use `Time` objects to figure it out:

```
Time startTime = new Time(18, 50, 0.0);
Time runningTime = new Time(2, 16, 0.0);
```

Here are two ways we could “add” the `Time` objects:

- Write a static method that takes two `Time` objects as parameters.
- Write an instance method that gets invoked on one object and takes the other as a parameter.

To demonstrate the difference, we’ll do both. Here is the static method:

```
public static Time add(Time t1, Time t2) {
    Time sum = new Time();
    sum.hour = t1.hour + t2.hour;
    sum.minute = t1.minute + t2.minute;
    sum.second = t1.second + t2.second;
    return sum;
}
```

And here’s how we would invoke it:

```
Time endTime = Time.add(startTime, runningTime);
```

Here’s what it looks like as an instance method:

```
public Time add(Time t2) {
    Time sum = new Time();
    sum.hour = this.hour + t2.hour;
    sum.minute = this.minute + t2.minute;
    sum.second = this.second + t2.second;
    return sum;
}
```

And here's how we would invoke it:

```
Time endTime = startTime.add(runningTime);
```

Notice the differences:

- The static method has the keyword `static`; the instance method does not.
- The static method has two parameters, `t1` and `t2`. The instance method has one explicit parameter, `t1`, and the implicit parameter, `this`.
- We invoked the static method with the `Time` class; we invoked the instance method with the `startTime` object.

That's all there is to it. Static methods and instance methods do the same thing, and you can convert from one to the other with just a few changes.

However, there's a problem with both of these methods; they are not correct. The result from either method is 20:66, which is not a valid time.

If `second` exceeds 59, we have to *carry* into the minutes column, and if `minute` exceeds 59, we have to carry into hour.

Here is a better version of the instance method, `add`:

```
public Time add(Time t2) {
    Time sum = new Time();
    sum.hour = this.hour + t2.hour;
    sum.minute = this.minute + t2.minute;
    sum.second = this.second + t2.second;

    if (sum.second >= 60.0) {
        sum.second -= 60.0;
        sum.minute += 1;
    }
    if (sum.minute >= 60) {
        sum.minute -= 60;
        sum.hour += 1;
    }
    if (sum.hour >= 24) {
        sum.hour -= 24
    }
    return sum;
}
```

If hour exceeds 23, we subtract 24 hours, but there's no days attribute to carry into.

Vocabulary

class

In [Chapter 1](#), we defined a class as a collection of related methods. Now you know that a class is also a template for a new type of object.

instance

A member of a class. Every object is an instance of a class.

instantiate

Create a new instance of a class in the computer's memory.

instance variable

An attribute of an object; a nonstatic variable defined at the class level.

information hiding

The practice of making instance variables private to limit dependencies between classes.

constructor

A special method that initializes the instance variables of a newly constructed object.

shadowing

Occurs when a local variable or parameter has the same name as an attribute.

client

A class that uses objects defined in another class.

getter

A method that returns the value of an instance variable.

setter

A method that assigns a value to an instance variable.

override

To replace a default implementation of a method, such as `toString`.

instance method

A nonstatic method that has access to `this` and the instance variables.

identical

Two objects that are the same or have the same location in memory.

equivalent

Two objects that have the same value, as defined by the `equals` method.

Exercises

The code for this chapter is in the *ch11* directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

Exercise 11-1.

The implementation of `increment` in this chapter is not very efficient. Can you rewrite it so it doesn't use any loops?

Hint: Remember the remainder operator—it works with floating-point values too.

Exercise 11-2.

In the board game Scrabble, each tile contains a letter, which is used to spell words in rows and columns, and a score, which is used to determine the value of words. The point of this exercise is to practice the mechanical part of creating a new class definition.

1. Write a definition for a class named `Tile` that represents Scrabble tiles. The instance variables should include a character named `letter` and an integer named `value`.
2. Write a constructor that takes parameters named `letter` and `value`, and initializes the instance variables.
3. Write a method named `printTile` that takes a `Tile` object as a parameter and displays the instance variables in a reader-friendly format.
4. Write a `main` method that creates a `Tile` object with the letter `Z` and the value `10`, and then uses `printTile` to display the state of the object.
5. Implement the `toString` and `equals` methods for a `Tile`.
6. Create getters and setters for each of the attributes.

Exercise 11-3.

Write a class definition for `Date`, an object type that contains three integers: `year`, `month`, and `day`. This class should provide two constructors. The first should take no parameters and initialize a default date. The second should take parameters named `year`, `month` and `day`, and use them to initialize the instance variables.

Write a `main` method that creates a new `Date` object named `birthday`. The new object should contain your birth date. You can use either constructor.

Exercise 11-4.

A *rational number* is a number that can be represented as the ratio of two integers. For example, $2/3$ is a rational number, and you can think of 7 as a rational number with an implicit 1 in the denominator.

1. Define a class called `Rational`. A `Rational` object should have two integer instance variables that store the numerator and denominator.
2. Write a constructor that takes no arguments and sets the numerator to 0 and denominator to 1.
3. Write an instance method called `printRational` that displays a `Rational` object in a reasonable format.
4. Write a `main` method that creates a new object with type `Rational`, sets its instance variables to the values of your choice, and displays the object.
5. You now have a minimal testable program. Test it and, if necessary, debug it.
6. Write a `toString` method for `Rational` and test it using `println`.
7. Write a second constructor that takes two arguments and uses them to initialize the instance variables.
8. Write an instance method called `negate` that reverses the sign of a rational number. This method should be a modifier, so it should be `void`. Add lines to `main` to test the new method.
9. Write an instance method called `invert` that swaps the numerator and denominator. It should be a modifier. Add lines to `main` to test the new method.
10. Write an instance method called `toDouble` that converts the rational number to a `double` (floating-point number) and returns the result. This method is a pure method; it does not modify the object. As always, test the new method.
11. Write an instance method named `reduce` that reduces a rational number to its lowest terms by finding the greatest common divisor (GCD) of the numerator and denominator and dividing through. This method should be a pure method; it should not modify the instance variables of the object on which it is invoked.
Hint: Finding the GCD takes only a few lines of code. Search the web for “Euclidean algorithm”.
12. Write an instance method called `add` that takes a `Rational` number as an argument, adds it to `this`, and returns a new `Rational` object. There are several ways to add fractions. You can use any one you want, but you should make sure that the result of the operation is reduced so that the numerator and denominator have no common divisor (other than 1).

Arrays of Objects

During the next three chapters, we will develop programs that work with playing cards and decks of cards. Here is an outline of the road ahead:

- In this chapter, we define a `Card` class and write methods that work with cards and arrays of cards.
- In [Chapter 13](#), we define a `Deck` class that encapsulates an array of cards, and we write methods that operate on decks.
- In [Chapter 14](#), we introduce a way to define new classes that extend existing classes. Then we use `Card` and `Deck` to implement the game Crazy Eights.

There are 52 cards in a standard deck. Each card belongs to one of four suits and one of 13 ranks. The suits are Clubs, Diamonds, Hearts, and Spades. The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King.

If you are unfamiliar with traditional playing cards, now would be a good time to get a deck or read through [Wikipedia's "Standard 52-card deck" entry](#).

Card Objects

If we want to define a class to represent a playing card, it is pretty clear what the instance variables should be: `rank` and `suit`. It is not as obvious what types they should be.

One possibility is a `String` containing things like "Spade" for suits and "Queen" for ranks. A problem with this choice is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. By encode, we *don't* mean to encrypt or translate into a secret code. We mean to define a mapping between a sequence of numbers and the things we want to represent.

Here is a mapping for suits:

Clubs ↦ 0
Diamonds ↦ 1
Hearts ↦ 2
Spades ↦ 3

We use the mathematical symbol \mapsto to make it clear that these mappings are not part of the program. They are part of the program design, but they never appear explicitly in the code.

Each of the numerical ranks (2 through 10) maps to the corresponding integer. For the face cards, we can use the following:

Ace ↦ 1
Jack ↦ 11
Queen ↦ 12
King ↦ 13

With this encoding, the class definition for the `Card` type looks like this:

```
public class Card {
    private int rank;
    private int suit;

    public Card(int rank, int suit) {
        this.rank = rank;
        this.suit = suit;
    }
}
```

The instance variables are `private`: we can access them from inside this class, but not from other classes.

The constructor takes a parameter for each instance variable. To create a `Card` object, we use the `new` operator:

```
Card threeOfClubs = new Card(3, 0);
```

The result is a reference to a `Card` that represents the 3 of Clubs.

Card toString

When you create a new class, the first step is to declare the instance variables and write constructors. A good next step is to write `toString`, which is useful for debugging and incremental development.

To display `Card` objects in a way that humans can read easily, we need to *decode* the integer values as words. A natural way to do that is with an array of `Strings`. For example, we can create the array like this:

```
String[] suits = new String[4];
```

And then assign values to the elements:

```
suits[0] = "Clubs";  
suits[1] = "Diamonds";  
suits[2] = "Hearts";  
suits[3] = "Spades";
```

Or we can create the array and initialize the elements at the same time, as you saw in “[Displaying Arrays](#)” on page 98:

```
String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
```

The memory diagram in [Figure 12-1](#) shows the result. Each element of the array is a reference to a `String`.

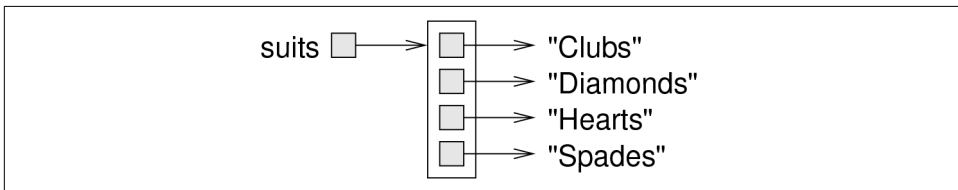


Figure 12-1. Memory diagram of an array of strings

We also need an array to decode the ranks:

```
String[] ranks = {null, "Ace", "2", "3", "4", "5", "6",  
"7", "8", "9", "10", "Jack", "Queen", "King"};
```

The zeroth element should never be used, because the only valid ranks are 1–13. We set it to `null` to indicate an unused element.

Using these arrays, we can create a meaningful `String` by using `suit` and `rank` as indexes.

```
String s = ranks[this.rank] + " of " + suits[this.suit];
```

The expression `ranks[this.rank]` means “use the instance variable `rank` from `this` object as an index into the array `ranks`.” We select the string for `this.suit` in a similar way.

Now we can wrap all the previous code in a `toString` method:

```
public String toString() {
    String[] ranks = {null, "Ace", "2", "3", "4", "5", "6",
                     "7", "8", "9", "10", "Jack", "Queen", "King"};
    String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
    String s = ranks[this.rank] + " of " + suits[this.suit];
    return s;
}
```

When we display a card, `println` automatically calls `toString`. The output of the following code is `Jack of Diamonds`:

```
Card card = new Card(11, 1);
System.out.println(card);
```

Class Variables

So far, you have seen local variables, which are declared inside a method, and instance variables, which are declared in a class definition, usually before the method definitions. Now it's time to learn about **class variables**. They are shared across all instances of the class.

Like instance variables, class variables are defined in a class definition, before the method definitions. But they are identified by the keyword `static`. Here is a version of `Card` in which `RANKS` and `SUITS` are defined as class variables:

```
public class Card {

    public static final String[] RANKS = {
        null, "Ace", "2", "3", "4", "5", "6", "7",
        "8", "9", "10", "Jack", "Queen", "King"};

    public static final String[] SUITS = {
        "Clubs", "Diamonds", "Hearts", "Spades"};

    // instance variables and constructors go here

    public String toString() {
        return RANKS[this.rank] + " of " + SUITS[this.suit];
    }
}
```

Class variables are allocated when the program begins and persist until the program ends. In contrast, instance variables like `rank` and `suit` are allocated when the program creates new objects, and they are deleted when the object is garbage-collected.

Class variables are often used to store constant values that are needed in several places. In that case, they should also be declared as `final`. Note that whether a variable is `static` or `final` involves two separate considerations: `static` means the variable is *shared*, and `final` means the variable is *constant*.

Naming `static final` variables with capital letters is a common convention that makes it easier to recognize their role in the class. In the `toString` method, we refer to `SUITS` and `RANKS` as if they were local variables, but we can tell that they are class variables.

One advantage of defining `SUITS` and `RANKS` as class variables is that they don't need to be created (and garbage-collected) every time `toString` is called. They may also be needed in other methods and classes, so it's helpful to make them available everywhere. Since the array variables are `final`, and the strings they reference are immutable, there is no danger in making them `public`.

The `compareTo` Method

As you saw in “[The equals Method](#)” on page 168, it's helpful to create an `equals` method to test whether two objects are equivalent:

```
public boolean equals(Card that) {  
    return this.rank == that.rank  
        && this.suit == that.suit;  
}
```

It would also be nice to have a method for comparing cards, so we can tell if one is higher or lower than another. For primitive types, we can use comparison operators like `<` and `>` to compare values. But these operators don't work for object types.

For strings, Java provides a `compareTo` method, as you saw in “[String Comparison](#)” on page 88. We can write our own version of `compareTo` for the classes that we define, as we did for the `equals` method.

Some types are *totally ordered*, which means that you can compare any two values and tell which is bigger. Integers and strings are totally ordered. Other types are *unordered*, which means that there is no meaningful way to say that one element is bigger than another. In Java, the `boolean` type is unordered; if you try to compare `true < false`, you get a compiler error.

The set of playing cards is *partially ordered*, which means that sometimes we can compare cards and sometimes not. For example, we know that the 3 of Clubs is higher than the 2 of Clubs, and the 3 of Diamonds is higher than the 3 of Clubs. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

To make cards comparable, we have to decide which is more important: rank or suit. The choice is arbitrary, and it might be different for different games. But when you buy a new deck of cards, it comes sorted with all the Clubs together, followed by all the Diamonds, and so on. So for now, let's say that suit is more important. With that decided, we can write `compareTo` as follows:

```
public int compareTo(Card that) {
    if (this.suit < that.suit) {
        return -1;
    }
    if (this.suit > that.suit) {
        return 1;
    }
    if (this.rank < that.rank) {
        return -1;
    }
    if (this.rank > that.rank) {
        return 1;
    }
    return 0;
}
```

`compareTo` returns `-1` if `this` is a lower card, `+1` if `this` is a higher card, and `0` if `this` and `that` are equivalent. It compares suits first. If the suits are the same, it compares ranks. If the ranks are also the same, it returns `0`.

Cards Are Immutable

The instance variables of `Card` are `private`, so they can't be accessed from other classes. We can provide getters to allow other classes to read the rank and suit values:

```
public int getRank() {
    return this.rank;
}

public int getSuit() {
    return this.suit;
}
```

Whether or not to provide setters is a design decision. If we did, cards would be mutable, so you could transform one card into another. That is probably not a feature we want, and in general, mutable objects are more error-prone. So it might be better to make cards immutable. To do that, all we have to do is *not* provide any modifier methods (including setters).

That's easy enough, but it is not foolproof, because a fool might come along later and add a modifier. We can prevent that possibility by declaring the instance variables `final`:

```

public class Card {
    private final int rank;
    private final int suit;
    ...
}

```

You can initialize these variables inside a constructor, but if someone writes a method that tries to modify them, they'll get a compiler error. This kind of safeguard helps prevent future mistakes and hours of debugging.

Arrays of Cards

Just as you can create an array of `String` objects, you can create an array of `Card` objects. The following statement creates an array of 52 cards. [Figure 12-2](#) shows the memory diagram for this array.

```
Card[] cards = new Card[52];
```

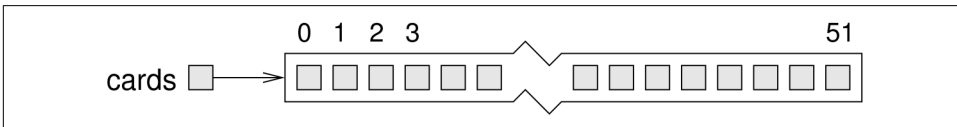


Figure 12-2. Memory diagram of an unpopulated `Card` array

Although we call it an *array of cards*, the array contains *references* to cards; it does not contain the `Card` objects themselves. Initially the references are all `null`.

Even so, you can access the elements of the array in the usual way:

```

if (cards[0] == null) {
    System.out.println("No card yet!");
}

```

But if you try to access the instance variables of nonexistent `Card` objects, you will get a `NullPointerException`:

```
System.out.println(cards[0].rank); // NullPointerException
```

That code won't work until we put cards in the array. One way to populate the array is to write nested for loops:

```

int index = 0;
for (int suit = 0; suit <= 3; suit++) {
    for (int rank = 1; rank <= 13; rank++) {
        cards[index] = new Card(rank, suit);
        index++;
    }
}

```

The outer loop iterates suits from 0 to 3. For each suit, the inner loop iterates ranks from 1 to 13. Since the outer loop runs 4 times, and the inner loop runs 13 times for each suit, the body is executed 52 times.

We use a separate variable `index` to keep track of where in the array the next card should go. [Figure 12-3](#) shows what the array looks like after the first two cards have been created.

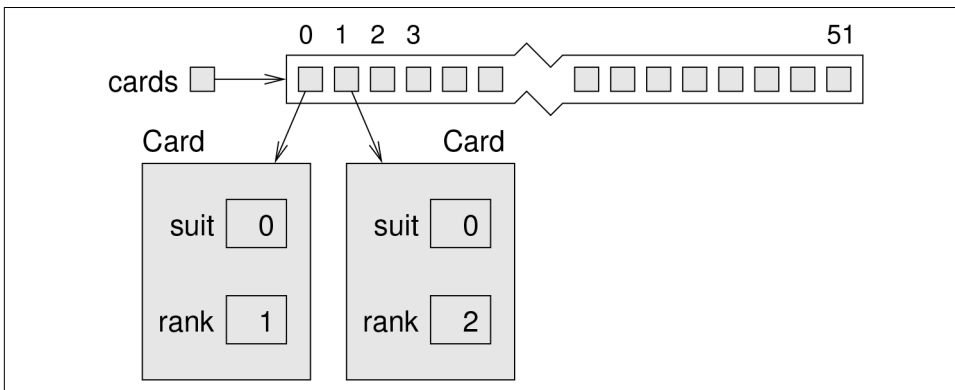


Figure 12-3. Memory diagram of a `Card` array with two cards

When you work with arrays, it is convenient to have a method that displays the contents. You have seen the pattern for traversing an array several times, so the following method should be familiar:

```
public static void printDeck(Card[] cards) {  
    for (Card card : cards) {  
        System.out.println(card);  
    }  
}
```

Since `cards` has type `Card[]`, pronounced “card array”, an element of `cards` has type `Card`. So `println` invokes the `toString` method in the `Card` class.

Then again, we don’t have to write our own `printDeck` method. The `Arrays` class provides a `toString` method that invokes `toString` on the elements of an array and concatenates the results:

```
System.out.println(Arrays.toString(cards))
```


Sequential Search

The next method we'll write is `search`, which takes an array of cards and a `Card` object as parameters. It returns the index where the `Card` appears in the array, or `-1` if it doesn't. This version of `search` uses the algorithm in “[Traversing Arrays](#)” on page 100, which is called **sequential search**:

```
public static int search(Card[] cards, Card target) {
    for (int i = 0; i < cards.length; i++) {
        if (cards[i].equals(target)) {
            return i;
        }
    }
    return -1;
}
```

The method returns as soon as it discovers the card, which means we don't have to traverse the entire array if we find the target. If we get to the end of the loop, we know the card is not in the array.

If the cards in the array are not in order, there is no way to search faster than sequential search. We have to look at every card, because otherwise we can't be certain the card we want is not there. But if the cards are in order, we can use better algorithms.

Sequential search is relatively inefficient, especially for large arrays. If you pay the price to keep the array sorted, finding elements becomes much easier.

Binary Search

When you look for a word in a dictionary, you don't search page by page from front to back. Since the words are in alphabetical order, you probably use a **binary search** algorithm:

1. Start on a page near the middle of the dictionary.
2. Compare a word on the page to the word you are looking for. If you find it, stop.
3. If the word on the page comes before the word you are looking for, flip to somewhere later in the dictionary and go to step 2.
4. If the word on the page comes after the word you are looking for, flip to somewhere earlier in the dictionary and go to step 2.

This algorithm is much faster than sequential search, because it rules out half of the remaining words each time you make a comparison. If at any point you find two adjacent words on the page, and your word comes between them, you can conclude that your word is not in the dictionary.

Getting back to the array of cards, we can write a faster version of search if we know the cards are in order:

```
public static int binarySearch(Card[] cards, Card target) {
    int low = 0;
    int high = cards.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;           // step 1
        int comp = cards[mid].compareTo(target);

        if (comp == 0) {                     // step 2
            return mid;
        } else if (comp < 0) {              // step 3
            low = mid + 1;
        } else {                             // step 4
            high = mid - 1;
        }
    }
    return -1;
}
```

First, we declare `low` and `high` variables to represent the range we are searching. Initially, we search the entire array, from `0` to `cards.length - 1`.

Inside the `while` loop, we repeat the four steps of binary search:

1. Choose an index between `low` and `high`—call it `mid`—and compare the card at `mid` to the target.
2. If you found the target, return its index (which is `mid`).
3. If the card at `mid` is lower than the target, search the range from `mid + 1` to `high`.
4. If the card at `mid` is higher than the target, search the range from `low` to `mid - 1`.

If `low` exceeds `high`, there are no cards in the range, so we terminate the loop and return `-1`.

This algorithm depends on only the `compareTo` method of the object, so we can use this code with any object type that provides `compareTo`.

Tracing the Code

To see how binary search works, it's helpful to add the following print statement at the beginning of the loop:

```
System.out.println(low + ", " + high);
```

Using a sorted deck of cards, we can search for the Jack of Clubs like this:

```
Card card = new Card(11, 0);
System.out.println(binarySearch(cards, card));
```

We expect to find this card at position 10 (since the Ace of Clubs is at position 0). Here is the output of `binarySearch`:

```
0, 51
0, 24
0, 11
6, 11
9, 11
10
```

You can see the range of cards shrinking as the `while` loop runs, until eventually index 10 is found. If we search for a card that's not in the array—like `new Card(15, 1)`, or the 15 of Diamonds—we get the following:

```
0, 51
26, 51
26, 37
26, 30
26, 27
-1
```

Each time through the loop, we cut the distance between `low` and `high` in half. After k iterations, the number of remaining cards is $52/2^k$. To find the number of iterations it takes to complete, we set $52/2^k = 1$ and solve for k . The result is $\log_2 52$, which is about 5.7. So we might have to look at 5 or 6 cards, as opposed to all 52 if we did a sequential search.

More generally, if the array contains n elements, binary search requires $\log_2 n$ comparisons, and sequential search requires n . For large values of n , binary search is substantially faster.

Vocabulary

encode

To represent one set of values using another set of values by constructing a mapping between them.

class variable

A variable declared within a class as `static`. There is only one copy of a class variable, no matter how many objects there are.

sequential search

An algorithm that searches array elements, one by one, until a target value is found.

binary search

An algorithm that searches a sorted array by starting in the middle, comparing an element to the target, and eliminating half of the remaining elements.

Exercises

The code for this chapter is in the *ch12* directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

Exercise 12-1.

Encapsulate the deck-building code from “Arrays of Cards” on page 181 in a method called `makeDeck` that takes no parameters and returns a fully populated array of `Cards`.

Exercise 12-2.

In some card games, Aces are ranked higher than Kings. Modify the `compareTo` method to implement this ordering.

Exercise 12-3.

In Poker a *flush* is a hand that contains five or more cards of the same suit. A hand can contain any number of cards.

1. Write a method called `suitHist` that takes an array of cards as a parameter and returns a histogram of the suits in the hand. Your solution should traverse the array only once, as in “Building a Histogram” on page 103.
2. Write a method called `hasFlush` that takes an array of cards as a parameter and returns `true` if the hand contains a flush (and `false` otherwise).

Exercise 12-4.

Working with cards is more fun if you can display them on the screen. If you have not already read [Appendix C](#) about 2D graphics, you should read it before working on this exercise. In the code directory for this chapter, *ch12*, you will find the following:

cardset-oxymoron

A directory containing images of playing cards.

CardTable.java

A sample program that demonstrates how to read and display images.

This code demonstrates the use of a 2D array; specifically, an array of images. The declaration looks like this:

```
private Image[][] images;
```

The variable `images` refers to a 2D array of `Image` objects, which are defined in the `java.awt` package. Here's the code that creates the array itself:

```
images = new Image[14][4];
```

The array has 14 rows (one for each rank plus an unused row for rank 0) and 4 columns (one for each suit). Here's the loop that populates the array:

```
String cardset = "cardset-oxymoron";
String suits = "cdhs";

for (int suit = 0; suit <= 3; suit++) {
    char c = suits.charAt(suit);

    for (int rank = 1; rank <= 13; rank++) {
        String s = String.format("%s/%02d%c.gif",
                                cardset, rank, c);
        images[rank][suit] = new ImageIcon(s).getImage();
    }
}
```

The variable `cardset` contains the name of the directory that contains the image files. `suits` is a string that contains the single-letter abbreviations for the suits. These strings are used to assemble `s`, which contains the filename for each image. For example, when `rank=1` and `suit=2`, the value of `s` is `"cardset-oxymoron/01h.gif"`, which is an image of the Ace of Hearts.

The last line of the loop reads the image file, extracts an `Image` object, and assigns it to a location in the array, as specified by the indexes `rank` and `suit`. For example, the image of the Ace of Hearts is stored in row 1, column 2.

If you compile and run `CardTable.java`, you should see images of a deck of cards laid out on a green table. You can use this class as a starting place to implement your own card games.

As a starting place, try placing cards on the table in the starting configuration for the solitaire game [Klondike](#).

You can get the image for the back of the card by reading the file `back192.gif`.

Objects of Arrays

In the previous chapter, we defined a class to represent cards and used an array of `Card` objects to represent a deck. In this chapter, we take additional steps toward object-oriented programming.

First we define a class to represent a deck of cards. Then we present algorithms for shuffling and sorting decks. Finally, we introduce `ArrayList` from the Java library and use it to represent collections of cards.

Decks of Cards

Here is the beginning of a `Deck` class that encapsulates an array of `Card` objects:

```
public class Deck {
    private Card[] cards;

    public Deck(int n) {
        this.cards = new Card[n];
    }

    public Card[] getCards() {
        return this.cards;
    }
}
```

The constructor initializes the instance variable with an array of `n` cards, but it doesn't create any `Card` objects. [Figure 13-1](#) shows what a `Deck` looks like with no cards.

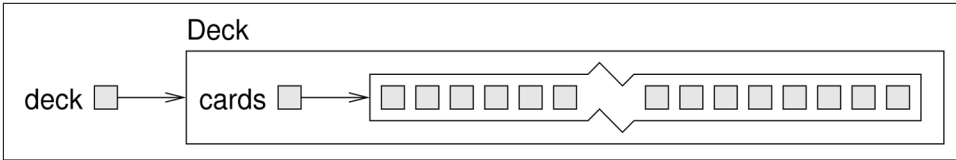


Figure 13-1. Memory diagram of an unpopulated Deck object

We'll add another constructor that creates a standard 52-card array and populates it with Card objects:

```
public Deck() {
    this.cards = new Card[52];
    int index = 0;
    for (int suit = 0; suit <= 3; suit++) {
        for (int rank = 1; rank <= 13; rank++) {
            this.cards[index] = new Card(rank, suit);
            index++;
        }
    }
}
```

This method is similar to the example in [“Arrays of Cards” on page 181](#); we just turned it into a constructor. We can use it to create a complete Deck like this:

```
Deck deck = new Deck();
```

Now that we have a Deck class, we have a logical place to put methods that pertain to decks. Looking at the methods we have written so far, one obvious candidate is `printDeck` from [“Arrays of Cards” on page 181](#). Here's how it looks, rewritten as an instance method of Deck:

```
public void print() {
    for (Card card : this.cards) {
        System.out.println(card);
    }
}
```

Notice that when we transform a static method into an instance method, the code is shorter. Here's how we invoke it:

```
deck.print();
```

Shuffling Decks

For most card games, you have to shuffle the deck; that is, put the cards in a random order. In [“Generating Random Numbers” on page 102](#), you saw how to generate random numbers, but it is not obvious how to use them to shuffle a deck.

One possibility is to model the way humans shuffle; for example, we could divide the deck in two halves and then choose alternately from each one. Humans usually don't shuffle perfectly, so after about seven iterations, the order of the deck is pretty well randomized.

But a computer program would have the annoying property of doing a perfect shuffle every time, which is not very random. In fact, after eight perfect shuffles, you would find the deck back in the order you started in! For more on this, see [Wikipedia's "Faro shuffle" entry](#).

A better shuffling algorithm is to traverse the deck one card at a time, and at each iteration, choose two cards and swap them. To outline this algorithm, we'll use a combination of Java statements and English comments. This technique is sometimes called **pseudocode**:

```
public void shuffle() {
    for each index i {
        // choose a random number between i and length - 1
        // swap the ith card and the randomly-chosen card
    }
}
```

The nice thing about pseudocode is that it often makes clear what other methods you are going to need. In this case, we need a method that chooses a random integer in a given range and a method that takes two indexes and swaps the cards at those positions:

```
private static int randomInt(int low, int high) {
    // return a random number between low and high,
    // including both
}

private void swapCards(int i, int j) {
    // swap the ith and the jth cards in the array
}
```

Methods like `randomInt` and `swapCards` are called **helper methods**, because they help you solve parts of the problem. Helper methods are often `private`, because they are used only by methods in the class and are not needed by methods in other classes.

The process of writing pseudocode first and then writing helper methods to make it work is a kind of **top-down design** see [Wikipedia's "Top-down and bottom-up design" entry](#). It is an alternative to *incremental development* and *encapsulation and generalization*, the other design processes you have seen in this book.

One of the exercises at the end of the chapter asks you to write the helper methods `randomInt` and `swapCards`, and use them to implement `shuffle`.

When you do the exercise, notice that `randomInt` is a class method and `swapCards` is an instance method. Do you understand why?

Selection Sort

Now that we have shuffled the deck, we need a way to put it back in order. There is an algorithm for sorting that is ironically similar to the algorithm for shuffling. It's called **selection sort**, because it works by traversing the array repeatedly and selecting the lowest (or highest) remaining card each time.

During the first iteration, we find the lowest card and swap it with the card in the zeroth position. During the i th iteration, we find the lowest card to the right of i and swap it with the i th card. Here is pseudocode for selection sort:

```
public void selectionSort() {
    for each index i {
        // find the lowest card at or to the right of i
        // swap the ith card and the lowest card found
    }
}
```

Again, the pseudocode helps with the design of the helper methods. For this algorithm, we can reuse `swapCards` from the previous section, so we need only a method to find the lowest card; we'll call it `indexLowest`:

```
private int indexLowest(int low, int high) {
    // find the lowest card between low and high
}
```

One of the exercises at the end of the chapter asks you to write `indexLowest`, and then use it and `swapCards` to implement `selectionSort`.

Merge Sort

Selection sort is a simple algorithm, but it is not very efficient. To sort n items, it has to traverse the array $n - 1$ times. Each traversal takes an amount of time proportional to n . The total time, therefore, is proportional to n^2 .

We will develop a more efficient algorithm called **merge sort**. To sort n items, merge sort takes time proportional to $n \log_2 n$. That may not seem impressive, but as n gets big, the difference between n^2 and $n \log_2 n$ can be enormous.

For example, \log_2 of one million is around 20. So if you had to sort a million numbers, merge sort would require 20 million steps. But selection sort would require one trillion steps!

The idea behind merge sort is this: if you have two decks, each of which has already been sorted, you can quickly merge them into a single, sorted deck. Try this out with a deck of cards:

1. Form two decks with about 10 cards each, and sort them so they are face up with the lowest cards on top. Place the decks in front of you.
2. Compare the top card from each deck and choose the lower one. Flip it over and add it to the merged deck.
3. Repeat step 2 until one of the decks is empty. Then take the remaining cards and add them to the merged deck.

The result should be a single sorted deck. In the next few sections, we'll explain how to implement this algorithm in Java.

Subdecks

The first step of merge sort is to split the deck into two *subdecks*, each with about half of the cards. So we need a method that takes a deck, and a range of indexes, and returns a new deck that contains the specified subset of cards:

```
public Deck subdeck(int low, int high) {
    Deck sub = new Deck(high - low + 1);
    for (int i = 0; i < sub.cards.length; i++) {
        sub.cards[i] = this.cards[low + i];
    }
    return sub;
}
```

The first line creates an unpopulated `Deck` object that contains an array of `null` references. Inside the `for` loop, the subdeck gets populated with references to `Card` objects.

The length of the subdeck is `high - low + 1`, because both the low card and the high card are included. This sort of computation can be confusing, and forgetting the “+ 1” often leads to **off-by-one** errors. Drawing a picture is usually the best way to avoid them.

Figure 13-2 is a memory diagram of a subdeck with `low = 0` and `high = 4`. The result is a hand with five cards that are *shared* with the original deck; that is, they are aliased.

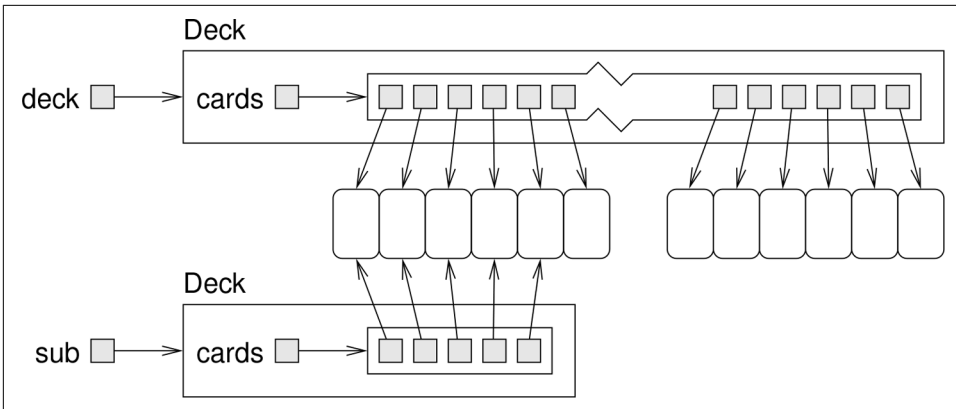


Figure 13-2. Memory diagram showing the effect of subdeck

Aliasing might not be a good idea, because changes to shared cards would be reflected in multiple decks. But since Card objects are immutable, this kind of aliasing is not a problem. And it saves some memory because we don't create duplicate Card objects.

Merging Decks

The next helper method we need is `merge`, which takes two sorted subdecks and returns a new deck containing all cards from both decks, in order. Here's what the algorithm looks like in pseudocode, assuming the subdecks are named `d1` and `d2`:

```
private static Deck merge(Deck d1, Deck d2) {
    // create a new deck, d3, big enough for all the cards

    // use the index i to keep track of where we are at in
    // the first deck, and the index j for the second deck
    int i = 0;
    int j = 0;

    // the index k traverses the result deck
    for (int k = 0; k < d3.length; k++) {
        // if d1 is empty, use top card from d2
        // if d2 is empty, use top card from d1
        // otherwise, compare the top two cards

        // add lowest card to the new deck at k
        // increment i or j (depending on card)
    }
    // return the new deck
}
```

An exercise at the end of the chapter asks you to implement `merge`. It's a little tricky, so be sure to test it with different subdecks. Once your `merge` method is working, you can use it to write a simplified version of merge sort:

```
public Deck almostMergeSort() {  
    // divide the deck into two subdecks  
    // sort the subdecks using selectionSort  
    // merge the subdecks, return the result  
}
```

If you have working versions of `subdeck`, `selectionSort`, and `merge`, you should have no trouble getting this method working. But it is still not very efficient, because it uses `selectionSort` to sort the subdecks. We can make it more efficient if we use `mergeSort` instead, but that means we have to make it recursive!

Adding Recursion

To make `mergeSort` work recursively, you have to add a base case; otherwise, it repeats forever. The simplest base case is a subdeck with one card. If there is only one card, it can't be out of order, so we consider it sorted. And if it is already sorted, we can just return it.

And it will turn out to be convenient if we handle another base case, a subdeck with zero cards. By the same logic, if there are no cards, they can't be out of order. So we consider an empty deck to be sorted, and return it.

With these base cases, a recursive version of `mergeSort` looks like this:

```
public Deck mergeSort() {  
    // if the deck has 0 or 1 cards, return it  
    // otherwise, divide the deck into two subdecks  
    // sort the subdecks using mergeSort  
    // merge the subdecks  
    // return the result  
}
```

As usual, there are two ways to think about recursive programs: you can follow the flow of execution, or you can make the leap of faith (see [“The Leap of Faith” on page 116](#)). This example should encourage you to make the leap of faith.

When you use `selectionSort` to sort the subdecks, you don't feel compelled to follow the flow of execution. You assume it works because you already debugged it. When you make `mergeSort` recursive, you just replace one sorting algorithm with another. There is no reason to read the program differently.

Well, almost. You have to think about the base cases and make sure that you reach them. But other than that, writing the recursive version should be no problem. As an exercise at the end of this chapter, you'll have a chance to finish off this example.

Static Context

Figure 13-3 shows a UML class diagram for Deck, including the instance variable, cards, and the methods we have so far. In UML diagrams, private attributes and methods begin with a minus sign (-) and static methods are underlined.

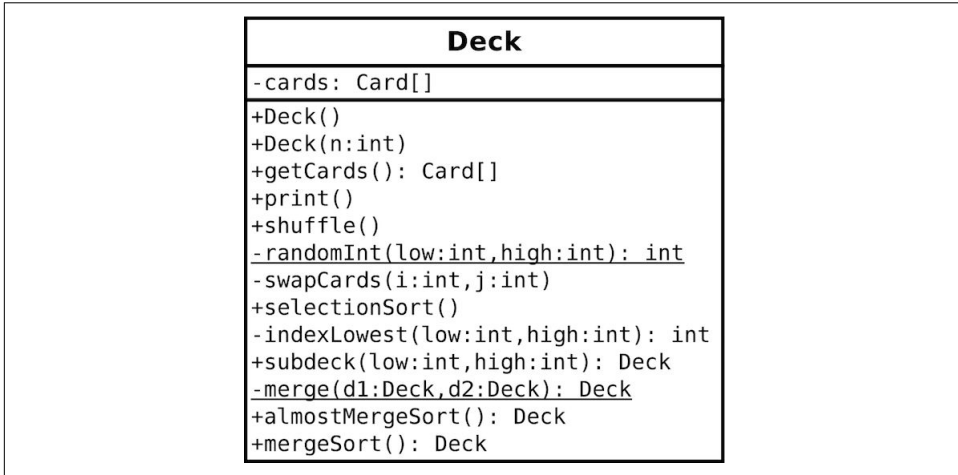


Figure 13-3. UML diagram for the Deck class

The helper methods `randomInt` and `merge` are static, because they do not read or write any instance variables. All other methods are instance methods, because they access the instance variable, `cards`.

When you have static methods and instance methods in the same class, it is easy to get them confused.

To invoke an instance method, you need an instance:

```
Deck deck = new Deck();
deck.print(); // correct
```

`Deck` with a capital D is a class, and `deck` with a lowercase d is an object.

Say you try to invoke `print` like this:

```
Deck.print(); // wrong!
```

You get a compiler error like this:

```
Non-static method print() cannot be referenced from a
static context.
```

By **static context**, the compiler means you are trying to invoke a method in a context that requires a static method.

On the other hand, if you have a `Deck` object, you can use it to invoke a static method:

```
Deck deck = new Deck();
int i = deck.randomInt(0, 51); // legal, but not good style
```

This is legal, but it is not considered good style, because someone reading this code would expect `randomInt` to be an instance method.

Another common error is to use `this` in a static method. For example, say you write something like this:

```
private static Deck merge(Deck d1, Deck d2) {
    return this.cards; // wrong!
}
```

You get a compiler error like this:

```
Non-static variable this cannot be referenced from a
static context.
```

The problem is that `cards` is an instance variable, so it is *nonstatic*, so you can't access it from a static method. In general, you can't use `this` in a static method, because a static method is not invoked on an object.

For beginners, error messages about nonstatic context can be confusing and frustrating. We hope this section helps.

Piles of Cards

Now that we have classes that represent cards and decks, let's use them to make a game. One of the simplest card games that children play is called **War**.

Initially, the deck is divided evenly into two piles, one for each player. During each round, each player takes the top card from their pile and places it, face up, in the center. Whoever has the highest-ranking card, ignoring suit, takes the two cards and adds them to the bottom of their pile. The game continues until one player has won the entire deck.

We could use the `Deck` class to represent the individual piles. However, our implementation of `Deck` uses a `Card` array, and the length of an array can't change. As the game progresses, we need to be able to add and remove cards from the piles.

We can solve this problem with an `ArrayList`, which is in the `java.util` package. An `ArrayList` is a **collection**, which is an object that contains other objects. It provides methods to add and remove elements, and it grows and shrinks automatically.

We define a new class named `Pile` to represent a pile of cards. It uses an `ArrayList` to store `Card` objects:

```
public class Pile {
    private ArrayList<Card> cards;

    public Pile() {
        this.cards = new ArrayList<Card>();
    }
}
```

When you declare an `ArrayList`, you specify the type it contains in angle brackets (<>). This declaration says that `cards` is not just an `ArrayList`; it's an `ArrayList` of `Card` objects. The constructor initializes `this.cards` with an empty `ArrayList`.

Now let's think about the methods we need to play the game. At the beginning of each round, each player draws a card from the top of their pile. So we define a method to do that:

```
public Card popCard() {
    return this.cards.remove(0); // from the top of the pile
}
```

`popCard` removes the `Card` at the beginning of the `ArrayList`, which we think of as the top of the pile. Because we use `ArrayList.remove`, it automatically shifts the remaining cards to fill the gap.

At the end of each round, the winner adds cards to the bottom of their pile. So we define a method to do that:

```
public void addCard(Card card) {
    this.cards.add(card); // to the bottom of the pile
}
```

`ArrayList` provides a method, `add`, that adds an element to the end of the collection, which we think of as the bottom of the pile.

To know when to stop the game, we have to check if one of the piles is empty. Here's a method to do that:

```
public boolean isEmpty() {
    return this.cards.isEmpty();
}
```

So far, these methods don't do very much; they just invoke methods on the instance variable, `cards`. Methods like these are called **wrapper methods** because they wrap one method with another.

Finally, to start the game, we need to divide the deck into two equal parts. We can do that with `subdeck` from “Subdecks” on page 193 and a new method, `addDeck`:

```
public void addDeck(Deck deck) {
    for (Card card : deck.getCards()) {
        this.cards.add(card);
    }
}
```

`addDeck` takes a `Deck` object, loops through the cards, and adds them to the `Pile`. Notice that it does not remove the cards from the `Deck`, so the `Deck` and the `Pile` share cards. But that won't be a problem because cards are immutable.

Playing War

Now we can use `Deck` and `Pile` to implement the game. We'll start by creating a deck and shuffling:

```
Deck deck = new Deck();
deck.shuffle();
```

Then we divide the `Deck` into two piles:

```
Pile p1 = new Pile();
p1.addDeck(deck.subdeck(0, 25));

Pile p2 = new Pile();
p2.addDeck(deck.subdeck(26, 51));
```

The game itself is a loop that repeats until one of the piles is empty. At each iteration, we draw a card from each pile and compare their ranks:

```
while (!p1.isEmpty() && !p2.isEmpty()) {
    // pop a card from each pile
    Card c1 = p1.popCard();
    Card c2 = p2.popCard();

    // compare the cards
    int diff = c1.getRank() - c2.getRank();
    if (diff > 0) {
        p1.addCard(c1);
        p1.addCard(c2);
    } else if (diff < 0) {
        p2.addCard(c1);
        p2.addCard(c2);
    } else {
        // it's a tie
    }
}
```

If the two cards have the same rank, it's a tie. In that case, each player draws four more cards. Whoever has the higher fourth card takes all cards in play. If there's another tie, they draw another four cards, and so on.

One of the exercises at the end of this chapter asks you to implement the `else` block when there's a tie.

After the `while` loop ends, we display the winner based on which pile is not empty:

```
if (p2.isEmpty()) {
    System.out.println("Player 1 wins!");
} else {
    System.out.println("Player 2 wins!");
}
```

`ArrayList` provides many other methods that we didn't use for this example. Take a minute to read the documentation, which you can find by doing a web search for "Java ArrayList".

Vocabulary

pseudocode

A way of designing programs by writing rough drafts in a combination of English and Java.

helper method

A method that implements part of a more complex algorithm; often it is not particularly useful on its own.

top-down design

Breaking down a problem into subproblems, and solving each subproblem one at a time.

selection sort

A simple sorting algorithm that searches for the smallest or largest element n times.

merge sort

A recursive sorting algorithm that divides an array into two parts, sorts each part (using merge sort), and merges the results.

off-by-one

A common programming mistake that results in iterating one time too many, or too few.

static context

The parts of a class that run without reference to a specific instance of the class.

collection

A Java library class, like `ArrayList`, that represents a group of objects.

wrapper method

A method that calls another method without doing much additional work.

Exercises

The code for this chapter is in the `ch13` directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

Exercise 13-1.

Write a `toString` method for the `Deck` class. It should return a single string that represents the cards in the deck. When it’s printed, this string should display the same results as the `print` method in “Decks of Cards” on page 189.

Hint: You can use the `+` operator to concatenate strings, but it is not very efficient. Consider using `java.lang.StringBuilder` instead; see “StringBuilder Objects” on page 156.

Exercise 13-2.

The goal of this exercise is to implement the shuffling algorithm from this chapter.

1. In the repository for this book, you should find the file named `Deck.java`. Check that you can compile it in your environment.
2. Implement the `randomInt` method. You can use the `nextInt` method provided by `java.util.Random`, which you saw in “Generating Random Numbers” on page 102.

Hint: To avoid creating a `Random` object every time `randomInt` is invoked, consider defining a class variable.

3. Write a `swapCards` method that takes two indexes and swaps the cards at the given locations.
4. Fill in the `shuffle` method by using the algorithm in “Shuffling Decks” on page 190.

Exercise 13-3.

The goal of this exercise is to implement the sorting algorithms from this chapter. Use the `Deck.java` file from the previous exercise or create a new one from scratch.

1. Implement the `indexLowest` method. Use the `Card.compareTo` method to find the lowest card in a given range of the deck, from `lowIndex` to `highIndex`, including both.
2. Fill in `selectionSort` by using the algorithm in “[Selection Sort](#)” on page 192.
3. Using the pseudocode in “[Merge Sort](#)” on page 192, implement the `merge` method. The best way to test it is to build and shuffle a deck. Then use `subdeck` to form two small subdecks, and use selection sort to sort them. Finally, pass the two halves to `merge` and see if it works.
4. Fill in `almostMergeSort`, which divides the deck in half, then uses `selectionSort` to sort the two halves, and uses `merge` to create a new, sorted deck. You should be able to reuse code from the previous step.
5. Implement `mergeSort` recursively. Remember that `selectionSort` is a modifier and `mergeSort` is a pure method, which means that they get invoked differently:

```
deck.selectionSort();    // modifies an existing deck
deck = deck.mergeSort(); // replaces old deck with new
```

Exercise 13-4.

You can learn more about the sorting algorithms presented in this chapter at [Toptal's Sorting Algorithms Animations](#). This site provides explanations of the algorithms, along with animations that show how they work. It also includes an analysis of their efficiency.

For example, *insertion sort* is an algorithm that inserts elements into place, one at a time. Read about it on the website and play the animations. Then write a method named `insertionSort` that implements this algorithm.

One goal of this exercise is to practice top-down design. Your solution should use a helper method, named `insert`, that implements the inner loop of the algorithm. `insertionSort` should invoke this method $n - 1$ times.

Exercise 13-5.

Find and open the file *War.java* in the repository. The `main` method contains all the code from the last section of this chapter. Check that you can compile and run this code before proceeding.

The program is incomplete; it does not handle the case when two cards have the same rank. Finish implementing the `main` method, beginning at the line that says: `// it's a tie.`

When there's a tie, draw three cards from each pile and store them in a collection, along with the original two. Then draw one more card from each pile and compare them. Whoever wins the tie takes all ten of these cards.

If one pile does not have at least four cards, the game ends immediately. If a tie ends with a tie, draw three more cards, and so on.

Notice that this program depends on `Deck.shuffle`, so you might have to do [Exercise 13-2](#) first.

Extending Classes

In this chapter, we present a comprehensive example of object-oriented programming. Crazy Eights is a classic card game for two or more players. The main objective is to be the first player to get rid of all your cards. Here's how to play:

- Deal five or more cards to each player, and then deal one card face up to create the *discard pile*. Place the remaining cards face down to create the *draw pile*.
- Each player takes turns placing a single card on the discard pile. The card must match the rank or suit of the previously played card, or be an eight, which is a *wildcard*.
- When players don't have a matching card or an eight, they must draw new cards until they get one.
- If the draw pile ever runs out, the discard pile is shuffled (except the top card) and becomes the new draw pile.
- As soon as a player has no cards, the game ends, and all other players score penalty points for their remaining cards. Eights are worth 20, face cards are worth 10, and all others are worth their rank.

You can read [Wikipedia's "Crazy Eights" entry](#) for more details, but we have enough to get started.

CardCollection

To implement Crazy Eights, we need to represent a deck of cards, a discard pile, a draw pile, and a hand for each player. And we need to be able to deal, draw, and discard cards.

The Deck and Pile classes from the previous chapter meet some of these requirements. But unless we make some changes, neither of them represents a hand of cards very well.

Furthermore, Deck and Pile are essentially two versions of the same code: one based on arrays, and the other based on ArrayList. It would be helpful to combine their features into one class that meets the needs of both.

We will define a class named CardCollection and add the code we want one step at a time. Since this class will represent different piles and hands of cards, we'll add a label attribute to tell them apart:

```
public class CardCollection {  
  
    private String label;  
    private ArrayList<Card> cards;  
  
    public CardCollection(String label) {  
        this.label = label;  
        this.cards = new ArrayList<Card>();  
    }  
}
```

As with the Pile class, we need a way to add cards to the collection. Here is the add Card method from the previous chapter:

```
public void addCard(Card card) {  
    this.cards.add(card);  
}
```

Until now, we have used this explicitly to make it easy to identify attributes. Inside addCard and other instance methods, you can access instance variables without using the keyword this. So from here on, we will drop it:

```
public void addCard(Card card) {  
    cards.add(card);  
}
```

We also need to be able to remove cards from the collection. The following method takes an index, removes the card at that location, and shifts the following cards left to fill the gap:

```
public Card popCard(int i) {  
    return cards.remove(i);  
}
```


If we are dealing cards from a shuffled deck, we don't care which card gets removed. It is most efficient to choose the last one, so we don't have to shift any cards left. Here is an overloaded version of `popCard` that removes and returns the last card:

```
public Card popCard() {
    int i = cards.size() - 1;    // from the end of the list
    return popCard(i);
}
```

`CardCollection` also provides `isEmpty`, which returns true if there are no cards left:

```
public boolean isEmpty() {
    return cards.isEmpty()
}
```

To access the elements of an `ArrayList`, you can't use the array `[]` operator. Instead, you have to use the methods `get` and `set`. Here is a wrapper for `get`:

```
public Card getCard(int i) {
    return cards.get(i);
}
```

`lastCard` gets the last card (but doesn't remove it):

```
public Card lastCard() {
    int i = size() - 1;
    return cards.get(i);
}
```

In order to control the ways card collections are modified, we don't provide a wrapper for `set`. The only modifiers we provide are the two versions of `popCard` and the following version of `swapCards`:

```
public void swapCards(int i, int j) {
    Card temp = cards.get(i);
    cards.set(i, cards.get(j));
    cards.set(j, temp);
}
```

Finally, we use `swapCards` to implement `shuffle`, which we described in “[Shuffling Decks](#)” on page 190:

```
public void shuffle() {
    Random random = new Random();
    for (int i = size() - 1; i > 0; i--) {
        int j = random.nextInt(i + 1);
        swapCards(i, j);
    }
}
```

Inheritance

At this point, we have a class that represents a collection of cards. It provides functionality common to decks of cards, piles of cards, hands of cards, and potentially other collections.

However, each kind of collection will be slightly different. Rather than add every possible feature to `CardCollection`, we can use **inheritance** to define subclasses. A **subclass** is a class that *extends* an existing class; that is, it has the attributes and methods of the existing class, plus more.

Here is the complete definition of our new and improved `Deck` class:

```
public class Deck extends CardCollection {  
  
    public Deck(String label) {  
        super(label);  
        for (int suit = 0; suit <= 3; suit++) {  
            for (int rank = 1; rank <= 13; rank++) {  
                addCard(new Card(rank, suit));  
            }  
        }  
    }  
}
```

The first line uses the keyword `extends` to indicate that `Deck` extends the class `CardCollection`. That means a `Deck` object has the same instance variables and methods as a `CardCollection`. Another way to say the same thing is that `Deck` *inherits from* `CardCollection`. We could also say that `CardCollection` is a **superclass**, and `Deck` is one of its subclasses.

In Java, classes may extend only one superclass. Classes that do not specify a superclass with `extends` automatically inherit from `java.lang.Object`. So in this example, `Deck` extends `CardCollection`, which in turn extends `Object`. The `Object` class provides the default `equals` and `toString` methods, among other things.

Constructors are *not* inherited, but all other `public` attributes and methods are. The only additional method in `Deck`, at least for now, is a constructor. So you can create a `Deck` object like this:

```
Deck deck = new Deck("Deck");
```

The first line of the constructor uses `super`, which is a keyword that refers to the superclass of the current class. When `super` is used as a method, as in this example, it invokes the constructor of the superclass.

So in this case, `super` invokes the `CardCollection` constructor, which initializes the attributes `label` and `cards`. When it returns, the `Deck` constructor resumes and populates the (empty) `ArrayList` with `Card` objects.

That's it for the Deck class. Next we need a way to represent a hand, which is the collection of cards held by a player, and a pile, which is a collection of cards on the table. We could define two classes, one for hands and one for piles, but there is not much difference between them. So we'll use one class, called Hand, for both hands and piles. Here's what the definition looks like:

```
public class Hand extends CardCollection {  
  
    public Hand(String label) {  
        super(label);  
    }  
  
    public void display() {  
        System.out.println(getLabel() + ": ");  
        for (int i = 0; i < size(); i++) {  
            System.out.println(getCard(i));  
        }  
        System.out.println();  
    }  
}
```

Like Deck, the Hand class extends CardCollection. So it inherits methods like `getLabel`, `size`, and `getCard`, which are used in `display`. Hand also provides a constructor, which invokes the constructor of CardCollection.

In summary, a Deck is just like a CardCollection, but it provides a different constructor. And a Hand is just like a CardCollection, but it provides an additional method, `display`.

Dealing Cards

To begin the game, we need to deal cards to each of the players. And during the game, we need to move cards between hands and piles. If we add the following method to CardCollection, it can meet both of these requirements:

```
public void deal(CardCollection that, int n) {  
    for (int i = 0; i < n; i++) {  
        Card card = popCard();  
        that.addCard(card);  
    }  
}
```

The `deal` method removes cards from the collection it is invoked on, `this`, and adds them to the collection it gets as a parameter, `that`. The second parameter, `n`, is the number of cards to deal. We will use this method to implement `dealAll`, which deals (or moves) all of the remaining cards:

```

public void dealAll(CardCollection that) {
    int n = size();
    deal(that, n);
}

```

At this point, we can create a Deck and start dealing cards. Here's a simple example that deals five cards to a hand, and deals the rest into a draw pile:

```

Deck deck = new Deck("Deck");
deck.shuffle();

Hand hand = new Hand("Hand");
deck.deal(hand, 5);
hand.display();

Hand drawPile = new Hand("Draw Pile");
deck.dealAll(drawPile);
System.out.printf("Draw Pile has %d cards.\n",
    drawPile.size());

```

Because the deck is shuffled randomly, you should get a different hand each time you run this example. The output will look something like this:

```

Hand:
5 of Diamonds
Ace of Hearts
6 of Clubs
6 of Diamonds
2 of Clubs

Draw Pile has 47 cards.

```

If you are a careful reader, you might notice something strange about this example. Take another look at the definition of `deal`. Notice that the first parameter is supposed to be a `CardCollection`. But we invoked it like this:

```

Hand hand = new Hand("Hand");
deck.deal(hand, 5);

```

The argument is a `Hand`, not a `CardCollection`. So why is this example legal?

It's because `Hand` is a subclass of `CardCollection`, so a `Hand` object is also considered to be a `CardCollection` object. If a method expects a `CardCollection`, you can give it a `Hand`, a `Deck`, or a `CardCollection`.

But it doesn't work the other way around: not every `CardCollection` is a `Hand`, so if a method expects a `Hand`, you have to give it a `Hand`, not a `CardCollection` or a `Deck`.

If it seems strange that an object can belong to more than one type, remember that this happens in real life too. Every cat is also a mammal, and every mammal is also an animal. But not every animal is a mammal, and not every mammal is a cat.

The Player Class

The Deck and Hand classes we have defined so far could be used for any card game; we have not yet implemented any of the rules specific to Crazy Eights. And that's probably a good thing, since it makes it easy to reuse these classes if we want to make another game in the future.

But now it's time to implement the rules. We'll use two classes: Player, which encapsulates player strategy, and Eights, which creates and maintains the state of the game. Here is the beginning of the Player definition:

```
public class Player {  
  
    private String name;  
    private Hand hand;  
  
    public Player(String name) {  
        this.name = name;  
        this.hand = new Hand(name);  
    }  
}
```

A Player has two private attributes: a name and a hand. The constructor takes the player's name as a string and saves it in an instance variable. In this example, we have to use this to distinguish between the instance variable and the parameter with the same name.

The primary method that Player provides is play, which decides which card to discard during each turn:

```
public Card play(Eights eights, Card prev) {  
    Card card = searchForMatch(prev);  
    if (card == null) {  
        card = drawForMatch(eights, prev);  
    }  
    return card;  
}
```

The first parameter is a reference to the Eights object that encapsulates the state of the game (coming up in the next section). The second parameter, prev, is the card on top of the discard pile.

play invokes two helper methods: searchForMatch and drawForMatch. Since we have not written them yet, this is an example of top-down development.

Here's searchForMatch, which looks in the player's hand for a card that matches the previously played card:

```
public Card searchForMatch(Card prev) {  
    for (int i = 0; i < hand.size(); i++) {  
        Card card = hand.getCard(i);  
    }  
}
```

```

        if (cardMatches(card, prev)) {
            return hand.popCard(i);
        }
    }
    return null;
}

```

The strategy is pretty simple: the for loop searches for the first card that's legal to play and returns it. If there are no cards that match, it returns null. In that case, we have to draw cards until we get a match, which is what `drawForMatch` does:

```

public Card drawForMatch(Eights eights, Card prev) {
    while (true) {
        Card card = eights.drawCard();
        System.out.println(name + " draws " + card);
        if (cardMatches(card, prev)) {
            return card;
        }
        hand.addCard(card);
    }
}

```

The while loop runs until it finds a match (we'll assume for now that it always finds one). The loop uses the `Eights` object to draw a card. If it matches, `drawForMatch` returns the card. Otherwise, it adds the card to the player's hand and repeats.

Both `searchForMatch` and `drawForMatch` use `cardMatches`, which is a static method, also defined in `Player`. This method is a straightforward translation of the rules of the game:

```

public static boolean cardMatches(Card card1, Card card2) {
    return card1.getSuit() == card2.getSuit()
        || card1.getRank() == card2.getRank()
        || card1.getRank() == 8;
}

```

Finally, `Player` provides a `score` method, which computes penalty points for cards left in a player's hand at the end of the game.

The Eights Class

In “[Shuffling Decks](#)” on page 190, we introduced top-down development. In this way of developing programs, we identify high-level goals, like shuffling a deck, and break them into smaller problems, like choosing a random element or swapping two elements.

In this section, we present **bottom-up design**, which goes the other way around: first we identify simple pieces we need and then we assemble them into more-complex algorithms.

Looking at the rules of Crazy Eights, we can identify some of the methods we'll need.

Now we can start implementing the pieces. Here is the beginning of the class definition for Eights, which encapsulates the state of the game:

```
public class Eights {  
  
    private Player one;  
    private Player two;  
    private Hand drawPile;  
    private Hand discardPile;  
    private Scanner in;  
}
```

In this version, there are always two players. One of the exercises at the end of the chapter asks you to modify this code to handle more players. The Eights class also includes a draw pile, a discard pile, and a Scanner, which we will use to prompt the user after each turn.

The constructor for Eights initializes the instance variables and deals the cards, similar to “Dealing Cards” on page 209. The next piece we'll need is a method that checks whether the game is over. If either hand is empty, we're done:

```
public boolean isDone() {  
    return one.getHand().isEmpty() || two.getHand().isEmpty();  
}
```

When the draw pile is empty, we have to shuffle the discard pile. Here is a method for that:

```
public void reshuffle() {  
    Card prev = discardPile.popCard();  
    discardPile.dealAll(drawPile);  
    discardPile.addCard(prev);  
    drawPile.shuffle();  
}
```

The first line saves the top card from discardPile. The next line transfers the rest of the cards to drawPile. Then we put the saved card back into discardPile and shuffle drawPile. We can use reshuffle as part of the draw method:

```
public Card drawCard() {  
    if (drawPile.isEmpty()) {  
        reshuffle();  
    }  
    return drawPile.popCard();  
}
```

The nextPlayer method takes the current player as a parameter and returns the player who should go next:

```

public Player nextPlayer(Player current) {
    if (current == one) {
        return two;
    } else {
        return one;
    }
}

```

The last method from our bottom-up design is `displayState`. It displays the hand of each player, the contents of the discard pile, and the number of cards in the draw pile. Finally, it waits for the user to press the Enter key:

```

public void displayState() {
    one.display();
    two.display();
    discardPile.display();
    System.out.println("Draw pile:");
    System.out.println(drawPile.size() + " cards");
    in.nextLine();
}

```

Using these pieces, we can write `takeTurn`, which executes one player's turn. It reads the top card off the discard pile and passes it to `player.play`, which you saw in the previous section. The result is the card the player chose, which is added to the discard pile:

```

public void takeTurn(Player player) {
    Card prev = discardPile.lastCard();
    Card next = player.play(this, prev);
    discardPile.addCard(next);

    System.out.println(player.getName() + " plays " + next);
    System.out.println();
}

```

Finally, we use `takeTurn` and the other methods to write `playGame`:

```

public void playGame() {
    Player player = one;

    // keep playing until there's a winner
    while (!isDone()) {
        displayState();
        takeTurn(player);
        player = nextPlayer(player);
    }

    // display the final score
    one.displayScore();
    two.displayScore();
}

```


Done! The result of bottom-up design is similar to top-down: we have a high-level method that calls helper methods. The difference is the development process we used to arrive at this solution.

Class Relationships

This chapter demonstrates two common relationships between classes:

Composition

Instances of one class contain references to instances of another class. For example, an instance of `Eights` contains references to two `Player` objects, two `Hand` objects, and a `Scanner`.

Inheritance

One class extends another class. For example, `Hand` extends `CardCollection`, so every instance of `Hand` is also a `CardCollection`.

Composition is also known as a **HAS-A** relationship, as in “`Eights` has a `Scanner`”. *Inheritance* is also known as an **IS-A** relationship, as in “a `Hand` is a `CardCollection`”. This vocabulary provides a concise way to talk about an object-oriented design.

There is also a standard way to represent these relationships graphically in UML class diagrams. As you saw in “[Class Diagrams](#)” on page 153, the UML representation of a class is a box with three sections: the class name, the attributes, and the methods. The latter two sections are optional when showing relationships.

Relationships between classes are represented by arrows: composition arrows have a standard arrow head, and inheritance arrows have a hollow triangle head (usually pointing up). [Figure 14-1](#) shows the classes defined in this chapter and the relationships among them.

UML is an international standard, so almost any software engineer in the world could look at this diagram and understand our design. And class diagrams are only one of many graphical representations defined in the UML standard.

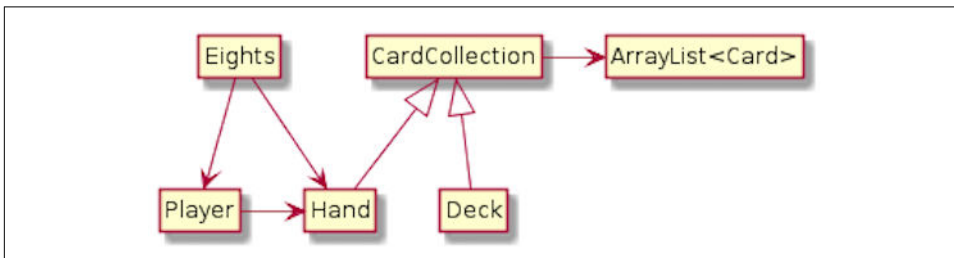


Figure 14-1. UML diagram for the classes in this chapter

Vocabulary

inheritance

The ability to define a new class that has the same instance variables and methods of an existing class.

subclass

A class that inherits from, or extends, an existing class.

superclass

An existing class that is extended by another class.

bottom-up design

A way of developing programs by identifying simple pieces, implementing them, and then assembling them into more-complex algorithms.

HAS-A

A relationship between two classes in which one class “has” an instance of another class as one of its attributes.

IS-A

A relationship between two classes in which one class extends another class; the subclass “is” an instance of the superclass.

Exercises

The code for this chapter is in the *ch14* directory of *ThinkJavaCode2*. See “[Using the Code Examples](#)” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

Exercise 14-1.

Design a better strategy for the `Player.play` method. For example, if there are multiple cards you can play, and one of them is an 8, you might want to play the 8.

Think of other ways you can minimize penalty points, such as playing the highest-ranking cards first. Write a new class that extends `Player` and overrides `play` to implement your strategy.

Exercise 14-2.

Write a loop that plays the game 100 times and keeps track of how many times each player wins. If you implemented multiple strategies in the previous exercise, you can play them against each other to evaluate which one works best.

Hint: Design a `Genius` class that extends `Player` and overrides the `play` method, and then replace one of the players with a `Genius` object.

Exercise 14-3.

One limitation of the program we wrote in this chapter is that it handles only two players. Modify the `Eights` class to create an `ArrayList` of players, and modify `nextPlayer` to select the next player.

Exercise 14-4.

When we designed the program for this chapter, we tried to minimize the number of classes. As a result, we ended up with a few awkward methods. For example, `cardMatches` is a static method in `Player`, but it would be more natural if it were an instance method in `Card`.

The problem is that `Card` is supposed to be useful for any card game, not just Crazy Eights. You can solve this problem by adding a new class, `EightsCard`, that extends `Card` and provides a method, `match`, that checks whether two cards match according to the rules of Crazy Eights.

At the same time, you could create a new class, `EightsHand`, that extends `Hand` and provides a method, `scoreHand`, that adds up the scores of the cards in the hand. And while you're at it, you could add a method named `scoreCard` to `EightsCard`.

Whether or not you actually make these changes, draw a UML class diagram that shows this alternative object hierarchy.

Arrays of Arrays

The last two chapters of this book use 2D graphics to illustrate more advanced object-oriented concepts. If you haven't yet read [Appendix C](#), you might want to read it now and become familiar with the `Canvas`, `Color`, and `Graphics` classes from the `java.awt` package. In this chapter, we use these classes to draw images and animations, and to run graphical simulations.

Conway's Game of Life

The Game of Life, or GoL for short, was developed by John Conway and popularized in 1970 in Martin Gardner's column in *Scientific American*. Conway calls it a *zero-player game* because no players are needed to choose strategies or make decisions. After you set up the initial conditions, you watch the game play itself. That turns out to be more interesting than it sounds; you can read about it at [Wikipedia's "Conway's Game of Life" entry](#).

The game board is a 2D grid of square cells. Each cell is either *alive* or *dead*; the color of the cell indicates its state. [Figure 15-1](#) shows an example grid configuration; the five black cells are alive.

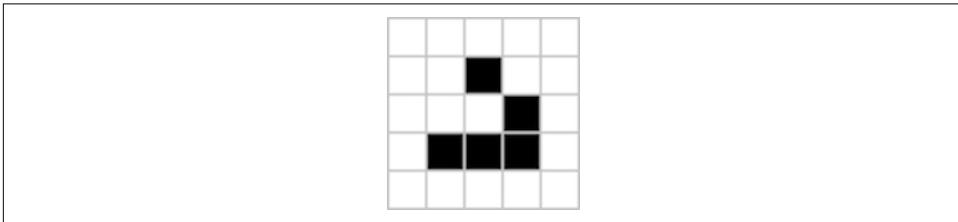


Figure 15-1. A Glider in the Game of Life

The game proceeds in time steps, during which each cell interacts with its neighbors in the eight adjacent cells. At each time step, the following rules are applied:

- A live cell with fewer than two live neighbors dies, as if by underpopulation.
- A live cell with more than three live neighbors dies, as if by overpopulation.
- A dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Notice some consequences of these rules. If you start with a single live cell, it dies. If all cells are dead, no cells come to life. But if you have four cells in a square, they keep each other alive, so that's a *stable* configuration.

Another initial configuration is shown in [Figure 15-2](#). If you start with three horizontal cells, the center cell lives, the left and right cells die, and the top and bottom cells come to life. The result after the first time step is three vertical cells.

During the next time step, the center cell lives, the top and bottom cells die, and the left and right cells come to life. The result is three horizontal cells, so we're back where we started, and the cycle repeats forever.

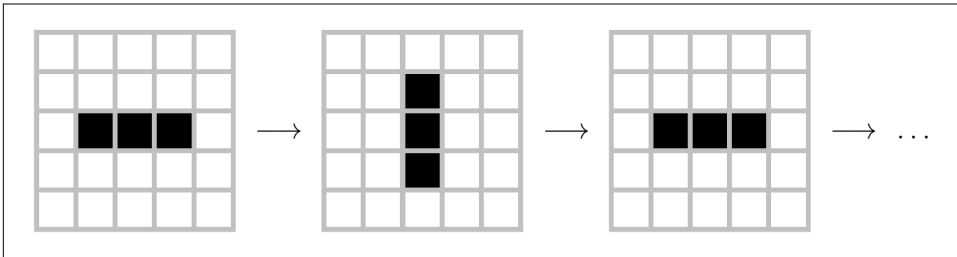


Figure 15-2. A Blinker in the Game of Life

Patterns like this are called *periodic*, because they repeat after a period of two or more time steps. But they are also considered *stable*, because the total number of live cells doesn't grow over time.

Most simple starting configurations either die out quickly or reach a stable configuration. But a few starting conditions display remarkable complexity. One of those is the **R-pentomino**: it starts with only five cells, runs for 1,103 time steps, and ends in a stable configuration with 116 live cells.

In the following sections, we'll implement the Game of Life in Java. We'll first implement the cells, then the grid of cells, and finally the game itself.

The Cell Class

When drawing a cell, we need to know its location on the screen and size in pixels. To represent the location, we use the `x` and `y` coordinates of the upper-left corner. And to represent the size, we use an integer, `size`.

To represent the state of a cell, we use an integer, `state`, which is 0 for dead cells and 1 for live cells. We could use a `boolean` instead, but it's good practice to design classes to be reusable (e.g., for other games that have more states).

Here is a `Cell` class that declares these instance variables:

```
public class Cell {
    private final int x;
    private final int y;
    private final int size;
    private int state;
}
```

Notice that `x`, `y`, and `size` are constants. Once the cell is created, we don't want it to move or change size. But `state` can and should change, so it is not a constant.

The next step is to write a constructor. Here's one that takes `x`, `y`, and `size` as parameters, and sets `state` to a default value:

```
public Cell(int x, int y, int size) {
    this.x = x;
    this.y = y;
    this.size = size;
    this.state = 0;
}
```

The following method draws a cell. Like the `paint` method in [Appendix C](#), it takes a graphics context as a parameter:

```
public static final Color[] COLORS = {Color.WHITE, Color.BLACK};

public void draw(Graphics g) {
    g.setColor(COLORS[state]);
    g.fillRect(x + 1, y + 1, size - 1, size - 1);
    g.setColor(Color.LIGHT_GRAY);
    g.drawRect(x, y, size, size);
}
```

The `draw` method uses the state of the cell to select a color from an array of `Color` objects. Then it uses `fillRect` to draw the center of the cell and `drawRect` to draw a light-gray border.

We also need methods to get and set the cell's state. We could just provide `getState` and `setState`, but the code will be more readable if we provide methods customized for the Game of Life:

```

public boolean isOff() {
    return state == 0;
}

public boolean isOn() {
    return state == 1;
}

public void turnOff() {
    state = 0;
}

public void turnOn() {
    state = 1;
}

```

Two-Dimensional Arrays

To represent a grid of cells, we can use a **multidimensional array**. To create an array, we specify the number of rows and columns:

```

int rows = 4;
int cols = 3;
Cell[][] array = new Cell[rows][cols];

```

The result is an array with four rows and three columns. Initially, the elements of the array are null. We can fill the array with Cell objects like this:

```

for (int r = 0; r < rows; r++) {
    int y = r * size;
    for (int c = 0; c < cols; c++) {
        int x = c * size;
        array[r][c] = new Cell(x, y, size);
    }
}

```

The loop variables *r* and *c* are the row and column indexes of the cells, respectively. The variables *x* and *y* are the coordinates. For example, if *size* is 10 pixels, the cell at index (1, 2) would be at coordinates (10, 20) on the screen.

In Java, a 2D array is really an array of arrays. You can think of it as an array of rows, where each row is an array. [Figure 15-3](#) shows what it looks like.

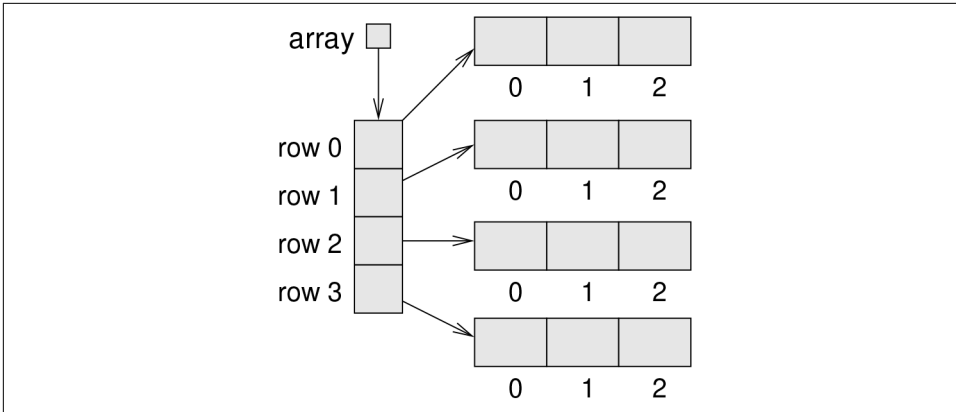


Figure 15-3. Storing rows and columns with a 2D array

When we write `array[r][c]`, Java uses the first index to select a row and the second index to select an element from the row. This way of representing 2D data is known as **row-major order**.

The GridCanvas Class

Now that we have a `Cell` class and a way to represent a 2D array of cells, we can write a class to represent a grid of cells. We encapsulate the code from the previous section and generalize it to construct a grid with any number of rows and columns:

```
public class GridCanvas extends Canvas {
    private Cell[][] array;

    public GridCanvas(int rows, int cols, int size) {
        array = new Cell[rows][cols];
        for (int r = 0; r < rows; r++) {
            int y = r * size;
            for (int c = 0; c < cols; c++) {
                int x = c * size;
                array[r][c] = new Cell(x, y, size);
            }
        }

        // set the canvas size
        setSize(cols * size, rows * size);
    }
}
```

Using vocabulary from the previous chapter, `GridCanvas` “is a” `Canvas` that “has a” 2D array of cells. By extending the `Canvas` class from `java.awt`, we inherit methods for drawing graphics on the screen.

In fact, the code is surprisingly straightforward: to draw the grid, we simply draw each cell. We use nested for loops to traverse the 2D array:

```
public void draw(Graphics g) {
    for (Cell[] row : array) {
        for (Cell cell : row) {
            cell.draw(g);
        }
    }
}
```

The outer loop traverses the rows; the inner loop traverses the cells in each row. You can almost read this method in English: “For each row in the array, and for each cell in the row, draw the cell in the graphics context.” Each cell contains its coordinates and size, so it knows how to draw itself.

Classes that extend `Canvas` are supposed to provide a method called `paint` that *paints* the contents of the `Canvas`. It gets invoked when the `Canvas` is created and anytime it needs to be redrawn; for example, when its window is moved or resized.

Here’s the `paint` method for `GridCanvas`. When the window management system calls `paint`, `paint` calls `draw`, which draws the cells:

```
public void paint(Graphics g) {
    draw(g);
}
```

Other Grid Methods

In addition to `draw` and `paint`, the `Grid` class provides methods for working with the grid itself. `numRows` and `numCols` return the number of rows and columns. We can get this information from the 2D array, using `length`:

```
public int numRows() {
    return array.length;
}

public int numCols() {
    return array[0].length;
}
```

Because we are using row-major order, the 2D array is an array of rows. `numRows` simply returns the length of the rows array. `numCols` returns the length of the first row, which is the number of columns. Since the rows all have the same length, we have to check only one.

`GridCanvas` also provides a method that gets the `Cell` at a given location, and for convenience when starting the game, a method that turns on the `Cell` at a given location:

```

public Cell getCell(int r, int c) {
    return array[r][c];
}

public void turnOn(int r, int c) {
    array[r][c].turnOn();
}

```

Starting the Game

Now we're ready to implement the game. To encapsulate the rules of GoL, we define a class named `Conway`. The `Conway` class "has a" `GridCanvas` that represents the state of the game.

This constructor makes a `GridCanvas` with 5 rows and 10 columns, with cells that are 20 pixels wide and high. It then sets up the initial conditions:

```

public class Conway {
    private GridCanvas grid;

    public Conway() {
        grid = new GridCanvas(5, 10, 20);
        grid.turnOn(2, 1);
        grid.turnOn(2, 2);
        grid.turnOn(2, 3);
        grid.turnOn(1, 7);
        grid.turnOn(2, 7);
        grid.turnOn(3, 7);
    }
}

```

Before we implement the rest of the game, we'll write a `main` method that creates a `Conway` object and displays it. We can use this method to test `Cell` and `GridCanvas`, and to develop the other methods we need:

```

public static void main(String[] args) {
    String title = "Conway's Game of Life";
    Conway game = new Conway();
    JFrame frame = new JFrame(title);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setResizable(false);
    frame.add(game.grid);
    frame.pack();
    frame.setVisible(true);
    game.mainloop();
}

```

After constructing the game object, `main` constructs a `JFrame`, which creates a window on the screen. The `JFrame` is configured to exit the program when closed. Resizing the window is disabled.

`main` then adds the `GridCanvas` inside the frame, resizes (*packs*) the frame to fit the canvas, and makes the frame visible. [Figure 15-4](#) shows the result.

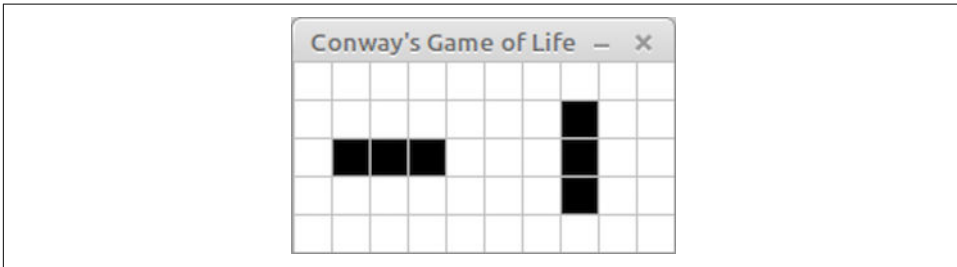


Figure 15-4. The initial Conway application

The Simulation Loop

At the end of `main`, we call `mainloop`, which uses a `while` loop to simulate the time steps of the Game of Life. Here's a rough draft of this method:

```
private void mainloop() {
    while (true) {
        this.update();
        grid.repaint();
        Thread.sleep(500); // compiler error
    }
}
```

During each time step, we update the state of the game and repaint the grid. We will present the update method in [“Updating the Grid” on page 229](#).

`repaint` comes from the `Canvas` class. By default, it calls the `paint` method we provided, which calls `draw`. The reason we use it here is that `repaint` does not require a `Graphics` object as a parameter.

`Thread.sleep(500)` causes the program to *sleep* for 500 milliseconds, or a half second. Otherwise, the program would run so fast we would not be able to see the animation.

There's just one problem: compiling this code results in the error `unreported exception InterruptedException`. This message means we need to do some exception handling.

Exception Handling

So far, the only exceptions you have seen are run-time errors like `array index out of bounds` and `null pointer`. When one of these exceptions occurs, Java displays a message and ends the program.

If you don't want the program to end, you can handle exceptions with a try-catch statement. The syntax is similar to an `if-else` statement, and the logic is too. Here's what it looks like:

```
try {
    Thread.sleep(500);
} catch (InterruptedException e) {
    // do nothing
}
```

First, Java runs the code in the try block, which calls `Thread.sleep` in this example. If an `InterruptedException` occurs during the try block, Java executes the catch block. In this example, the catch block contains a comment, so it doesn't do anything.

If a different exception occurs during the try block, Java does whatever it would do otherwise, which is probably to display a message and end the program. If no exceptions occur during the try block, the catch block doesn't run and the program continues.

In this example, the effect of the try-catch statement is to ignore an “interrupted” exception if it occurs. As an alternative, we could use the catch block to display a customized message, end the program, or handle the exception in whatever way is appropriate. For example, if user input causes an exception, we could catch the exception and prompt the user to try again later.

There's more to learn about exception handling. You can read about exceptions in the [Java tutorials](#).

Counting Neighbors

Now that you know about try and catch, we can use them to implement a useful method in `GridCanvas`. Part of the GoL logic is to count the number of live neighbors. Most cells have eight neighbors, as shown in [Figure 15-5](#).

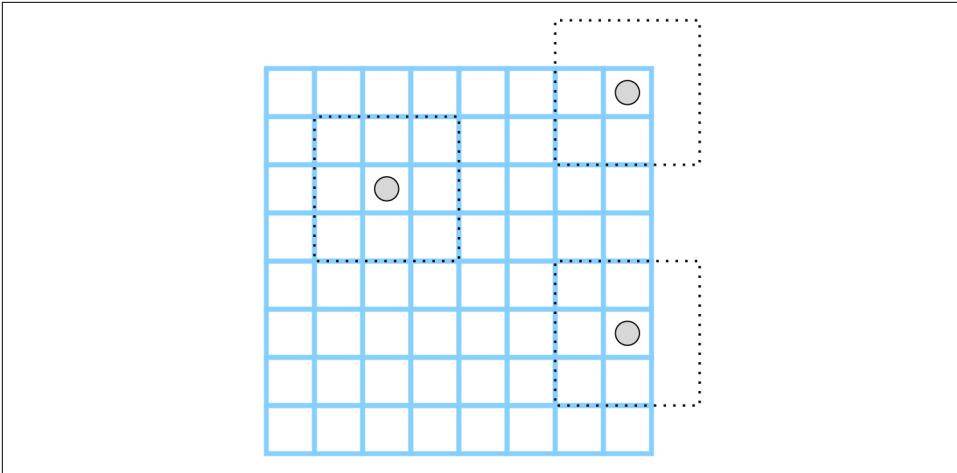


Figure 15-5. Cells in the interior of the grid have eight neighbors. Cells in the corners and along the edges have fewer neighbors.

However, cells on the edges and in the corners have fewer neighbors. If we try to count all possible neighbors, we'll go out of bounds. The following method uses a try-catch statement to deal with these special cases:

```
public int test(int r, int c) {
    try {
        if (array[r][c].isOn()) {
            return 1;
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        // cell doesn't exist
    }
    return 0;
}
```

The `test` method takes a row index, `r`, and a column index, `c`. It tries to look up the `Cell` at that location. If both indexes are in bounds, the `Cell` exists. In that case, `test` returns 1 if the `Cell` is on. Otherwise, it skips the catch block and returns 0.

If either index is out of bounds, the array lookup throws an exception, but the catch block ignores it. Then `test` resumes and returns 0. So the nonexistent cells around the perimeter are considered to be off.

Now we can use `test` to implement `countAlive`, which takes a grid location, (r, c) , and returns the number of live neighbors surrounding that location:

```

private int countAlive(int r, int c) {
    int count = 0;
    count += grid.test(r - 1, c - 1);
    count += grid.test(r - 1, c);
    count += grid.test(r - 1, c + 1);
    count += grid.test(r, c - 1);
    count += grid.test(r, c + 1);
    count += grid.test(r + 1, c - 1);
    count += grid.test(r + 1, c);
    count += grid.test(r + 1, c + 1);
    return count;
}

```

Because `test` handles *out-of-bounds* exceptions, `countAlive` works for any values of `r` and `c`.

Updating the Grid

Now we are ready to write `update`, which gets invoked each time through the simulation loop. It uses the GoL rules to compute the state of the grid after the next time step:

```

public void update() {
    int[][] counts = countNeighbors();
    updateGrid(counts);
}

```

The rules of GoL specify that you have to update the cells *simultaneously*; that is, you have to count the neighbors for all cells before you can update any of them.

We do that by traversing the grid twice: first, `countNeighbors` counts the live neighbors for each cell and puts the results in an array named `counts`; second, `updateGrid` updates the cells. Here's `countNeighbors`:

```

private int[][] countNeighbors() {
    int rows = grid.numRows();
    int cols = grid.numCols();

    int[][] counts = new int[rows][cols];
    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++) {
            counts[r][c] = countAlive(r, c);
        }
    }
    return counts;
}

```

`countNeighbors` traverses the cells in the grid and uses `countAlive` from the previous section to count the neighbors. The return value is a 2D array of integers with the same size as `grid`. [Figure 15-6](#) illustrates an example.

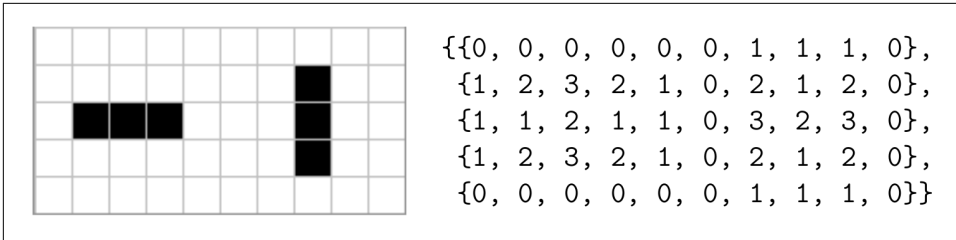


Figure 15-6. The result of countNeighbors

In contrast to the draw method of GridCanvas, which uses enhanced for loops, countNeighbors uses standard for loops. The reason is that, in this example, we need the indexes `r` and `c` to store the neighbor counts.

updateGrid uses `getCell` to select each Cell in the grid, and `updateCell` to do the update:

```

private void updateGrid(int[][] counts) {
    int rows = grid.numRows();
    int cols = grid.numCols();

    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++) {
            Cell cell = grid.getCell(r, c);
            updateCell(cell, counts[r][c]);
        }
    }
}

```

updateCell implements the GoL rules: if the cell is alive, it dies if it has fewer than two or more than three neighbors; if the cell is dead, it comes to life if it has exactly three:

```

private static void updateCell(Cell cell, int count) {
    if (cell.isOn()) {
        if (count < 2 || count > 3) {
            cell.turnOff();
        }
    } else {
        if (count == 3) {
            cell.turnOn();
        }
    }
}

```

Notice that `updateGrid` and `updateCell` are both private, because they are helper methods not intended to be invoked from outside the class. `updateCell` is also static, because it does not depend on `grid`.

Now our implementation of the Game of Life is complete. We think it's pretty fun, and we hope you agree. But more importantly, this example is meant to demonstrate the use of 2D arrays and an object-oriented design that's a little more substantial than in previous chapters.

Vocabulary

multidimensional array

An array with more than one dimension; 2D arrays are *arrays of arrays*.

row-major order

Storing data in a 2D array, first by rows and then by columns.

Exercises

The code for this chapter is in the *ch15* directory of *ThinkJavaCode2*. See “[Using the Code Examples](#)” on [page xii](#) for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

Exercise 15-1.

In `GridCanvas`, write a method named `countOn` that returns the total number of cells that are *on*. This method can be used, for example, to track the population in Game of Life over time.

Exercise 15-2.

In our version of the Game of Life, the grid has a finite size. As a result, moving objects such as Gliders either crash into the wall or go out of bounds.

An interesting variation of the Game of Life is a *toroidal* grid, meaning that the cells *wrap around* on the edges. Modify the `test` method of `GridCanvas` so that the coordinates `r` and `c` map to the opposite side of the grid if they are too low or too high.

Run your code with a Glider (see [Figure 15-1](#)) to see if it works. You can initialize the Glider by modifying the constructor in the `Conway` class or by reading it from a file (see the next exercise).

Exercise 15-3.

The [LifeWiki site](#) has a fascinating collection of patterns for the Game of Life. These patterns are stored in a file format that is easy to read, in files with the suffix *.cells*.

For example, here is an 8×10 grid with a Glider near the upper-left corner:

```
!Name: Glider
.....
..0.....
...0.....
.000.....
.....
.....
.....
.....
```

Lines that begin with ! are comments and should be ignored. The rest of the file describes the grid, row by row. A period represents a dead cell, and an uppercase O represents a live cell. See [the LifeWiki's Plaintext page](#) for more examples.

1. Create a plain-text file with the preceding contents, and save the file as *glider.cells* in the same directory as your code.
2. Define a constructor for the Conway class that takes a string representing the name (or path) of a *.cells* file. Here is a starting point:

```
public Conway(String path) {
    File file = new File(path);
    Scanner scan = new Scanner(file);
}
```

3. Modify the main method to invoke the constructor as follows:

```
Conway game = new Conway("glider.cells");
```

4. Handle the `FileNotFoundException` that may be thrown when creating a `Scanner` for a `File` by invoking `printStackTrace` on the exception object and calling `System.exit` with a status of 1, indicating an error.
5. Continue implementing the constructor by reading all noncomment lines into an `ArrayList` via `hasNextLine` and `nextLine` of the `Scanner`.
6. Determine the number of rows and columns of the grid by examining the `ArrayList` contents.
7. Create and initialize a `GridCanvas` based on the `ArrayList`.

Once your constructor is working, you will be able to run many of the patterns on the LifeWiki. You might want to add a margin of empty cells around the initial pattern, to give it room to grow.

Exercise 15-4.

Some files on the LifeWiki use *run-length encoding (RLE)* instead of plain text. The basic idea of RLE is to describe the number of dead and alive cells, rather than type out each individual cell.

For example, *glider.cells* from the previous exercise could be represented this way with RLE:

```
#C Name: Glider
x = 10, y = 8
$2bo$3bo$b3o!
```

The first line specifies *x* (the number of columns) and *y* (the number of rows). Subsequent lines consist of the letters *b* (dead), *o* (alive), and *\$* (end of line), optionally preceded by a count. The pattern ends with *!*, after which any remaining file contents are ignored.

Lines beginning with *#* have special meaning and are not part of the pattern. For example, *#C* is a comment line. You can read more about [RLE format](#) on the LifeWiki site.

1. Create a plain-text file with the preceding contents, and save the file as *glider.rle* in the same directory as your code.
2. Modify your constructor from the previous exercise to check the last three characters of the path. If they are "rle", then you will need to process the file as RLE. Otherwise, assume the file is in *.cells* format.
3. In the end, your constructor should be able to read and initialize grids in both formats. Test your constructor by modifying the `main` method to read different files.

Reusing Classes

In [Chapter 15](#), we developed classes to implement Conway’s Game of Life. We can reuse the `Cell` and `GridCanvas` classes to implement other simulations. One of the most interesting zero-player games is *Langton’s Ant*, which models an “ant” that walks around a grid. The ant follows only two simple rules:

1. If the ant is on a white cell, it turns to the right, makes the cell black, and moves forward.
2. If the ant is on a black cell, it turns to the left, makes the cell white, and moves forward.

Because the rules are simple, you might expect the ant to do something simple, like make a square or repeat a simple pattern. But starting on a grid with all white cells, the ant makes more than 10,000 steps in a seemingly random pattern before it settles into a repeating loop of 104 steps. You can read more about it at: [Wikipedia’s “Langton’s Ant” entry](#).

In this chapter, we present a solution to Langton’s Ant and use it to demonstrate more advanced object-oriented techniques.

Langton’s Ant

We begin by defining a `Langton` class that has a grid and information about the ant. The constructor takes the grid dimensions as parameters:

```
public class Langton {
    private GridCanvas grid;
    private int xpos;
    private int ypos;
    private int head; // 0=North, 1=East, 2=South, 3=West
```

```

public Langton(int rows, int cols) {
    grid = new GridCanvas(rows, cols, 10);
    xpos = rows / 2;
    ypos = cols / 2;
    head = 0;
}
}

```

grid is a GridCanvas object, which represents the state of the cells. xpos and ypos are the coordinates of the ant, and head is the *heading* of the ant; that is, which direction it is facing. head is an integer with four possible values, where 0 means the ant is facing “north” (i.e., toward the top of the screen), 1 means “east”, etc. Here’s an update method that implements the rules for Langton’s Ant:

```

public void update() {
    flipCell();
    moveAnt();
}

```

The flipCell method gets the Cell at the ant’s location, figures out which way to turn, and changes the state of the cell:

```

private void flipCell() {
    Cell cell = grid.getCell(xpos, ypos);
    if (cell.isOff()) {
        head = (head + 1) % 4;    // turn right
        cell.turnOn();
    } else {
        head = (head + 3) % 4;    // turn left
        cell.turnOff();
    }
}
}

```

We use the remainder operator, %, to make head wrap around: if head is 3 and we turn right, it wraps around to 0; if head is 0 and we turn left, it wraps around to 3. Notice that to turn right, we add 1 to head. To turn left, we could subtract 1, but -1 % 4 is -1 in Java. So we add 3 instead, since one left turn is the same as three right turns.

The moveAnt method moves the ant forward one square, using head to determine which way is forward:

```

private void moveAnt() {
    if (head == 0) {
        ypos -= 1;
    } else if (head == 1) {
        xpos += 1;
    } else if (head == 2) {
        ypos += 1;
    } else {
        xpos -= 1;
    }
}
}

```

Here is the `main` method we use to create and display the `Langton` object:

```
public static void main(String[] args) {
    String title = "Langton's Ant";
    Langton game = new Langton(61, 61);
    JFrame frame = new JFrame(title);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setResizable(false);
    frame.add(game.grid);
    frame.pack();
    frame.setVisible(true);
    game.mainloop();
}
```

Most of this code is the same as the `main` we used to create and run `Conway`, in “[Starting the Game](#)” on page 225. It creates and configures a `JFrame` and runs `mainloop`.

And that’s everything! If you run this code with a grid size of 61×61 or larger, you will see the ant eventually settle into a repeating pattern.

Because we designed `Cell` and `GridCanvas` to be reusable, we didn’t have to modify them at all. However, we now have two copies of the `main` method—one on `Conway`, and one in `Langton`.

Refactoring

Whenever you see repeated code like `main`, you should think about ways to remove it. In [Chapter 14](#), we used inheritance to eliminate repeated code. We’ll do something similar with `Conway` and `Langton`.

First, we define a superclass named `Automaton`, in which we will put the code that `Conway` and `Langton` have in common:

```
public class Automaton {
    private GridCanvas grid;

    public void run(String title, int rate) {
        JFrame frame = new JFrame(title);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setResizable(false);
        frame.add(this.grid);
        frame.pack();
        frame.setVisible(true);
        this.mainloop(rate);
    }
}
```

`Automaton` declares `grid` as an instance variable, so every `Automaton` “has a” `GridCanvas`. It also provides `run`, which contains the code that creates and configures the `JFrame`.

The run method takes two parameters: the window title and the frame rate; that is, the number of time steps to show per second. It uses title when creating the JFrame, and it passes rate to mainloop:

```
private void mainloop(int rate) {
    while (true) {

        // update the drawing
        this.update();
        grid.repaint();

        // delay the simulation
        try {
            Thread.sleep(1000 / rate);
        } catch (InterruptedException e) {
            // do nothing
        }
    }
}
```

mainloop contains the code you first saw in “The Simulation Loop” on page 226. It runs a while loop forever (or until the window closes). Each time through the loop, it runs update to update grid and then repaint to redraw the grid.

Then it calls Thread.sleep with a delay that depends on rate. For example, if rate is 2, we should draw two frames per second, so the delay is a half second, or 500 milliseconds.

This process of reorganizing existing code, without changing its behavior, is known as **refactoring**. We’re almost finished; we just need to redesign Conway and Langton to extend Automaton.

Abstract Classes

So far, the implementation works, and if we are not planning to implement any other zero-person games, we could leave well enough alone. But there are a few problems with the current design:

- The grid attribute is private, making it inaccessible in Conway and Langton. We could make it public, but then other (unrelated) classes would have access to it as well.
- The Automaton class has no constructors, and even if it did, there would be no reason to create an instance of this class.
- The Automaton class does not provide an implementation of update. In order to work properly, subclasses need to provide one.

Java provides language features to solve these problems:

- We can make the `grid` attribute `protected`, which means it's accessible to subclasses but not other classes.
- We can make the class `abstract`, which means it cannot be instantiated. If you attempt to create an object for an abstract class, you will get a compiler error.
- We can declare `update` as an abstract method, meaning that it must be overridden in subclasses. If the subclass does not override an abstract method, you will get a compiler error.

Here's what `Automaton` looks like as an abstract class:

```
public abstract class Automaton {
    protected GridCanvas grid;

    public abstract void update();

    public void run(String title, int delay) {
        // body of this method omitted
    }

    private void mainloop(int rate) {
        // body of this method omitted
    }
}
```

Notice that the `update` method has no body. The declaration specifies the name, arguments, and return type. But it does not provide an implementation, because it is an abstract method.

Notice also the word `abstract` on the first line, which declares that `Automaton` is an abstract class. In order to have any abstract methods, a class must be declared as `abstract`.

Any class that extends `Automaton` must provide an implementation of `update`; the declaration here allows the compiler to check.

Here's what `Conway` looks like as a subclass of `Automaton`:

```
public class Conway extends Automaton {

    // same methods as before, except mainloop is removed

    public static void main(String[] args) {
        String title = "Conway's Game of Life";
        Conway game = new Conway("pulsar.cells", 2);
        game.run(title, 2);
    }
}
```

Conway extends Automaton, so it inherits the protected instance variable `grid` and the methods `run` and `mainLoop`. But because Automaton is abstract, Conway has to provide `update` and a constructor (which it has already).

Abstract classes are essentially *incomplete* class definitions that specify methods to be implemented by subclasses. But they also provide attributes and methods to be inherited, thus eliminating repeated code.

UML Diagram

At the beginning of the chapter, we had three classes: `Cell`, `GridCanvas`, and `Conway`. We then developed `Langton`, which had almost the same `main` method as `Conway`. So we refactored the code and created `Automaton`. Figure 16-1 summarizes the final design.

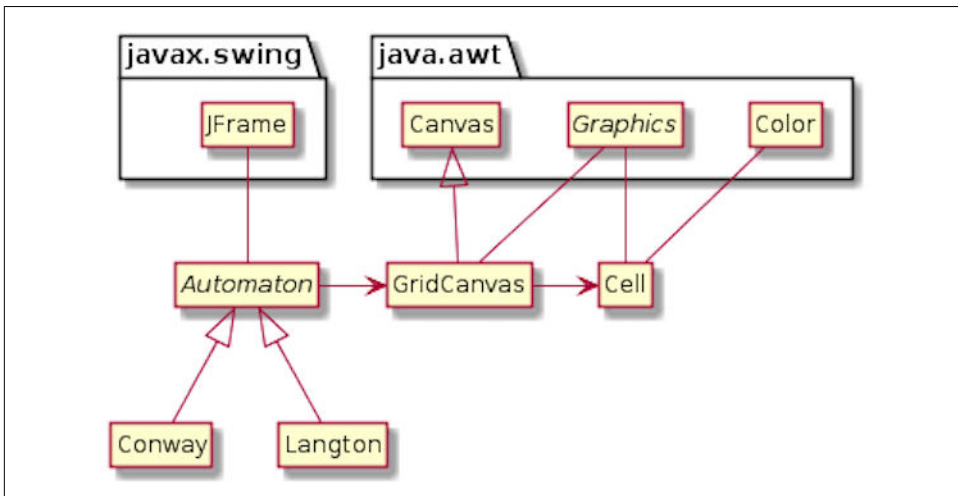


Figure 16-1. UML class diagram of Conway and Langton applications

The diagram shows three examples of inheritance: `Conway` is an `Automaton`, `Langton` is an `Automaton`, and `GridCanvas` is a `Canvas`. It also shows two examples of composition: `Automaton` has a `GridCanvas`, and `GridCanvas` has a 2D array of `Cells`.

The diagram also shows that `Automaton` uses `JFrame`, `GridCanvas` uses `Graphics`, and `Cell` uses `Graphics` and `Color`.

`Automaton` is in italics to indicate that it is an abstract class. As it happens, `Graphics` is an abstract class, too.

Conway and Langton are **concrete classes**, because they provide an implementation for all of their methods. In particular, they implement the update method that was declared abstract in `Automaton`.

One of the challenges of object-oriented programming is keeping track of a large number of classes and the relationships between them. UML class diagrams can help.

Vocabulary

refactor

To restructure or reorganize existing code without changing its behavior.

abstract class

A class that is declared as **abstract**; it cannot be instantiated, and it may (or may not) include abstract methods.

concrete class

A class that is *not* declared as **abstract**; each of its methods must have an implementation.

Exercises

Exercise 16-1.

The last section of this chapter introduced `Automaton` as an abstract class and rewrote Conway as a subclass of `Automaton`. Now it's your turn: rewrite Langton as a subclass of `Automaton`, removing the code that's no longer needed.

Exercise 16-2.

Mathematically speaking, Game of Life and Langton's Ant are **cellular automata**; *cellular* means it has cells, and *automaton* means it runs itself.

Implement another cellular automaton of your choice. You may have to modify `Cell` and/or `GridCanvas`, in addition to extending `Automaton`. For example, **Brian's Brain** requires three states: *on*, *dying*, and *off*.

Advanced Topics

When we first looked at inheritance in [Chapter 14](#), our purpose was to avoid duplicating code. We noticed that *decks of cards* and *hands of cards* had common functionality, and we designed a `CardCollection` class to provide it. This technique is an example of **generalization**. By generalizing the code, we were able to reuse it in the `Deck` and `Hand` classes.

In [Chapter 15](#), we looked at inheritance from a different point of view. When designing `GridCanvas` to represent a grid of cells, we extended `Canvas` and overrode its `paint` method. This design is an example of **specialization**. Using the code provided by `Canvas`, we created a specialized subclass with minimal additional code.

We didn't write the code for `Canvas`; it's part of the Java library. But we were able to customize it for our own purposes. In fact, the `Canvas` class was explicitly designed to be extended.

In this chapter, we'll explore the concept of inheritance more fully and explore event-driven programming. We'll continue to develop graphical simulations as a running example, but this time in varying shapes and colors!

Polygon Objects

The word *polygon* means *many angles*; the most basic polygons are triangles (three angles), rectangles (four angles), pentagons (five angles), and so forth. Polygons are an important part of computer graphics because they are used to compose more complex images.

Java provides a `Polygon` class (in `java.awt`) that we can use to represent and draw polygons. The following code creates an empty `Polygon` and adds three points, forming a triangle:

```
Polygon p = new Polygon();
p.addPoint(57, 110);
p.addPoint(100, 35);
p.addPoint(143, 110);
```

Internally, Polygon objects have three attributes:

- `public int npoints;`
 // total number of points
- `public int[] xpoints;`
 // array of X coordinates
- `public int[] ypoints;`
 // array of Y coordinates

When a Polygon is created, `npoints` is 0 and the two arrays are initialized with length 4. As points are added, `npoints` is incremented. If `npoints` exceeds the length of the arrays, larger arrays are created, and the previous values are copied over (similar to how ArrayList works).

The Polygon class provides many useful methods, like `contains`, `intersects`, and `translate`. We'll get to those later, but first we're going to do some specialization.

Adding Color

Specialization is useful for adding new features to an existing class, especially when you can't (or don't want to) change its design. For example, we can extend the Polygon class by adding a `draw` method and a `Color` attribute:

```
public class DrawablePolygon extends Polygon {
    public Color color;

    public DrawablePolygon() {
        super();
        color = Color.GRAY;
    }

    public void draw(Graphics g) {
        g.setColor(color);
        g.fillPolygon(this);
    }
}
```

As a reminder, constructors are not inherited when you extend a class. If you don't define a constructor, the compiler will generate one that does nothing.

The constructor for `DrawablePolygon` uses `super` to invoke the constructor for `Polygon`, which initializes the attributes `npoints`, `xpoints`, and `ypoints`. Then `DrawablePolygon` initializes the `color` attribute to `GRAY`.

`DrawablePolygon` has the same attributes and methods that `Polygon` has, so you can use `addPoint` as before, or you can directly access `npoints`, `xpoints`, and `ypoints` (since they are public). You can also use methods like `contains`, `intersects`, and `translate`.

The following code creates a `DrawablePolygon` with the same points as in the previous section and sets its color to `GREEN`:

```
DrawablePolygon p = new DrawablePolygon();
p.addPoint(57, 110);
p.addPoint(100, 35);
p.addPoint(143, 110);
p.color = Color.GREEN;
```

Regular Polygons

A *regular polygon* has all sides the same length and all angles equal in measure. Regular polygons are a special case of polygons, so we will use specialization to define a class for them.

We could extend the `Polygon` class, as we did in the previous section. But then we would not have the `Color` functionality we just added. So we will make `RegularPolygon` extend `DrawablePolygon`.

To construct a `RegularPolygon`, we specify the number of sides, the radius (distance from the center to a vertex), and the color. For example:

```
RegularPolygon rp = new RegularPolygon(6, 50, Color.BLUE);
```

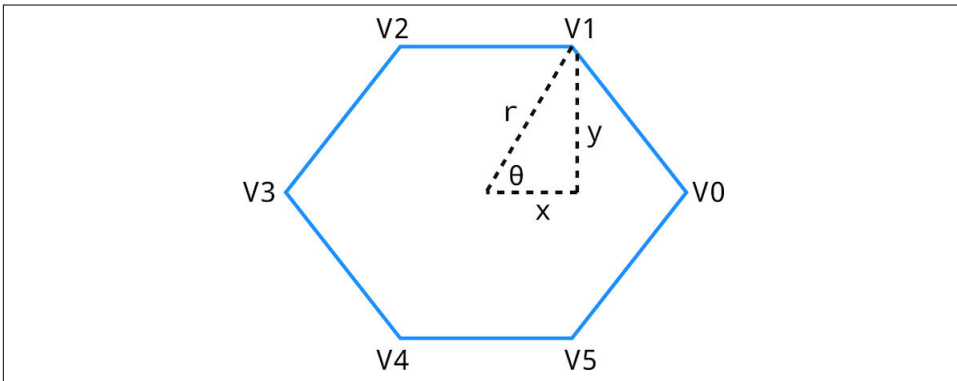


Figure 17-1. Determining the x and y coordinates of vertex $V1$, given the radius r and angle θ . The center of the polygon is at the origin $(0,0)$.

The constructor uses trigonometry to find the coordinates of each vertex. [Figure 17-1](#) illustrates the process. The number of sides ($n = 6$) and the radius ($r = 50$) are given as parameters.

- Imagine a clock hand starting at V_0 and rotating counterclockwise to V_1 , V_2 , and so forth. In [Figure 17-1](#), the hand is currently at V_1 .
- The angle θ is $2\pi/n$, since there are 2π radians in a circle. In other words, we are dividing the rotation of the clock hand into n equal angles.
- By definition, $\cos(\theta) = x/r$ and $\sin(\theta) = y/r$. Therefore, $x = r \cos(\theta)$ and $y = r \sin(\theta)$.
- We can determine the other (x, y) coordinates by multiplying θ by i , where i is the vertex number.

Here is the constructor for `RegularPolygon`:

```
public RegularPolygon(int nsides, int radius, Color color) {  
  
    // initialize DrawablePolygon attributes  
    this.npoints = nsides;  
    this.xpoints = new int[nsides];  
    this.ypoints = new int[nsides];  
    this.color = color;  
  
    // the amount to rotate for each vertex (in radians)  
    double theta = 2.0 * Math.PI / nsides;  
  
    // compute x and y coordinates, centered at the origin  
    for (int i = 0; i < nsides; i++) {  
        double x = radius * Math.cos(i * theta);  
        double y = radius * Math.sin(i * theta);  
        xpoints[i] = (int) Math.round(x);  
        ypoints[i] = (int) Math.round(y);  
    }  
}
```

This constructor initializes all four `DrawablePolygon` attributes, so it doesn't have to invoke `super()`.

It initializes `xpoints` and `ypoints` by creating arrays of integer coordinates. Inside the for loop, it uses `Math.sin` and `Math.cos` (see [“Math Methods” on page 51](#)) to compute the coordinates of the vertices as floating-point numbers. Then it rounds them off to integers and stores them in the arrays.

When we construct a `RegularPolygon`, the vertices are centered at the point $(0, 0)$. If we want the center of the polygon to be somewhere else, we can use `translate`, which we inherit from `Polygon`:


```
RegularPolygon rp = new RegularPolygon(6, 50, Color.BLUE);
rp.translate(100, 100);
```

The result is a six-sided polygon with radius 50 centered at the point (100, 100).

More Constructors

Classes in the Java library often have more than one constructor for convenience. We can do the same with `RegularPolygon`. For example, we can make the `color` parameter optional by defining a second constructor:

```
public RegularPolygon(int nsides, int radius) {
    this(nsides, radius, Color.GRAY);
}
```

The keyword `this`, when used in a constructor, invokes another constructor in the same class. It has a similar syntax as the keyword `super`, which invokes a constructor in the superclass.

Similarly, we could make the `radius` parameter optional too:

```
public RegularPolygon(int nsides) {
    this(nsides, 50);
}
```

Now, suppose we invoke the `RegularPolygon` constructor like this:

```
RegularPolygon rp = new RegularPolygon(6);
```

Because we provide only one integer argument, Java calls the third constructor, which calls the second one, which calls the first one. The result is a `RegularPolygon` with the specified value of `nsides`, 6, the default value of `radius`, 50, and the default color, `GRAY`.

When writing constructors, it's a good idea to validate the values you get as arguments. Doing so prevents run-time errors later in the program, which makes the code easier to debug.

For `RegularPolygon`, the number of sides should be at least three, the radius should be greater than zero, and the color should not be `null`. We can add the following lines to the first constructor:

```
public RegularPolygon(int nsides, int radius, Color color) {
    // validate the arguments
    if (nsides < 3) {
        throw new IllegalArgumentException("invalid nsides");
    }
    if (radius <= 0) {
        throw new IllegalArgumentException("invalid radius");
    }
}
```

```

if (color == null) {
    throw new NullPointerException("invalid color");
}

// the rest of the method is omitted

```

In this example, we throw an exception to indicate that one of the arguments is invalid. By default, these exceptions terminate the program and display an error message along with the stack trace.

Because we added this code to the most general constructor, we don't have to add it to the others.

An Initial Drawing

Now that we have `DrawablePolygon` and `RegularPolygon`, let's take them for a test drive. We'll need a `Canvas` for drawing them, so we define a new class, `Drawing`, that extends `Canvas`:

```

public class Drawing extends Canvas {
    private ArrayList<DrawablePolygon> list;

    public Drawing(int width, int height) {
        setSize(width, height);
        setBackground(Color.WHITE);
        list = new ArrayList<DrawablePolygon>();
    }

    public void add(DrawablePolygon cp) {
        list.add(cp);
    }

    public void paint(Graphics g) {
        for (DrawablePolygon dp : list) {
            dp.draw(g);
        }
    }
}

```

The `Drawing` class has an `ArrayList` of `DrawablePolygon` objects. When we create a `Drawing` object, the list is initially empty. The `add` method takes a `DrawablePolygon` and adds it to the list.

`Drawing` overrides the `paint` method that it inherits from `Canvas`. `paint` loops through the list of `DrawablePolygon` objects and invokes `draw` on each one.

Here is an example that creates three `RegularPolygon` objects and draws them. [Figure 17-2](#) shows the result.

```
public static void main(String[] args) {  
  
    // create some regular polygons  
    DrawablePolygon p1 = new RegularPolygon(3, 50, Color.GREEN);  
    DrawablePolygon p2 = new RegularPolygon(6, 50, Color.ORANGE);  
    DrawablePolygon p3 = new RegularPolygon(360, 50, Color.BLUE);  
  
    // move them out of the corner  
    p1.translate(100, 80);  
    p2.translate(250, 120);  
    p3.translate(400, 160);  
  
    // create drawing, add polygons  
    Drawing drawing = new Drawing(500, 250);  
    drawing.add(p1);  
    drawing.add(p2);  
    drawing.add(p3);  
  
    // set up the window frame  
    JFrame frame = new JFrame("Drawing");  
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    frame.add(drawing);  
    frame.pack();  
    frame.setVisible(true);  
}
```

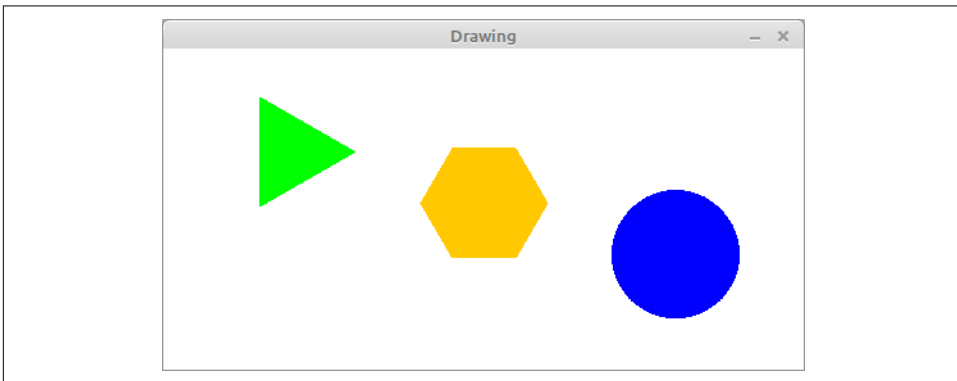


Figure 17-2. Initial drawing of three `RegularPolygon` objects

The first block of code creates `RegularPolygon` objects with 3, 6, and 360 sides. As you can see, a polygon with 360 sides is a pretty good approximation of a circle.

The second block of code translates the polygons to different locations. The third block of code creates the `Drawing` and adds the polygons to it. And the fourth block of code creates a `JFrame`, adds the `Drawing` to it, and displays the result.

Most of these pieces should be familiar, but one part of this program might surprise you. When we create the `RegularPolygon` objects, we assign them to `DrawablePolygon` variables. It might not be obvious why that's legal.

`RegularPolygon` extends `DrawablePolygon`, so every `RegularPolygon` object is also a `DrawablePolygon`. The parameter of `Drawing.add` has to be a `DrawablePolygon`, but it can be any type of `DrawablePolygon`, including `RegularPolygon` and other subclasses.

This design is an example of **polymorphism**, a fancy word that means *having many forms*. `Drawing.add` is a polymorphic method, because the parameter can be one of many types. And the `ArrayList` in `Drawing` is a polymorphic data structure, because the elements can be different types.

Blinking Polygons

At this point, we have a simple program that draws polygons; we can make it more fun by adding animation. [Chapter 15](#) introduced the idea of simulating time steps. Here's a loop that runs the animation:

```
while (true) {
    drawing.step();
    try {
        Thread.sleep(1000 / 30);
    } catch (InterruptedException e) {
        // do nothing
    }
}
```

Each time through the loop, we call `step` to update the `Drawing`. Then we sleep with a delay calculated to update about 30 times per second.

Here's what the `step` method of `Drawing` looks like:

```
public void step() {
    for (DrawablePolygon dp : list) {
        dp.step();
    }
    repaint();
}
```

It invokes `step` on each `DrawablePolygon` in the list and then repaints (clears and redraws) the canvas.

In order for this code to compile, we need `DrawablePolygon` to provide a `step` method. Here's a version that doesn't do anything; we'll override it in subclasses:

```
public void step() {
    // do nothing
}
```

Now let's design a new type of polygon that blinks. We'll define a class named `BlinkingPolygon` that extends `RegularPolygon` and adds two more attributes: `visible`, which indicates whether the polygon is visible, and `count`, which counts the number of time steps since the last blink:

```
public class BlinkingPolygon extends RegularPolygon {
    public boolean visible;
    public int count;

    public BlinkingPolygon(int nsides, int radius, Color c) {
        super(nsides, radius, c);
        visible = true;
        count = 0;
    }

    public void draw(Graphics g) {
        if (visible) {
            super.draw(g);
        }
    }

    public void step() {
        count++;
        if (count == 10) {
            visible = !visible;
            count = 0;
        }
    }
}
```

The constructor uses `super` to call the `RegularPolygon` constructor. Then it initializes `visible` and `count`. Initially, the `BlinkingPolygon` is visible.

The `draw` method draws the polygon only if it is visible. It uses `super` to call `draw` in the parent class. But the parent class is `RegularPolygon`, which does not provide a `draw` method. In this case, `super` invokes `draw` from the `DrawablePolygon` class.

The `step` method increments `count`. Every 10 time steps, it toggles `visible` and resets `count` to 0.

Interfaces

You might be getting tired of polygons at this point. Can't we draw anything else? Of course we can, but `Drawing` is currently based on `DrawablePolygon`. To draw other types of objects, we have to generalize the code.

The `Drawing` class does essentially three things: (1) it maintains a list of objects, (2) it invokes the `draw` method on each object, and (3) it invokes the `step` method on each object.

So here's one way we could make the code more general:

1. Define a new superclass, which we call `Actor`, that provides the two methods needed by `Drawing`:

```
public class Actor {
    public void draw(Graphics g) {
        // do nothing
    }
    public void step() {
        // do nothing
    }
}
```

2. In the `Drawing` class, replace `DrawablePolygon` with `Actor`.
3. Any class that we want to draw must now extend `Actor`.

There's just one problem: `DrawablePolygon` already extends `Polygon`, and classes can extend only one superclass. Also, the `Actor` class seems pointless, since the methods it defines don't do anything.

Java provides another mechanism for inheritance that solves these problems. We can define `Actor` as an interface instead of a class, like this:

```
public interface Actor {
    void draw(Graphics g);
    void step();
}
```

Like a class definition, an **interface** definition contains methods. But it contains only the declarations of the methods, not their implementations.

Like an abstract class, an interface specifies methods that must be provided by subclasses. The difference is that an abstract class can implement some methods; an interface cannot.

All interface methods are `public` by default, since they are intended to be used by other classes. So there is no need to declare them as `public`.

To inherit from an interface, you use the keyword `implements` instead of `extends`. Here's a version of `DrawablePolygon` that extends `Polygon` and implements `Actor`. So it inherits methods from `Polygon`, and it is required to provide the methods in `Actor`; namely, `step` and `draw`:

```
public class DrawablePolygon extends Polygon implements Actor {
    // rest of the class omitted
}
```

In terms of inheritance, `DrawablePolygon` is both a `Polygon` and an `Actor`. So the following assignments are legal:

```
Polygon p1 = new DrawablePolygon();
Actor a2 = new DrawablePolygon();
```

And the same is true for subclasses of `DrawablePolygon`; these assignments are legal too:

```
Polygon p2 = new RegularPolygon(5, 50, Color.YELLOW);
Actor a2 = new RegularPolygon(5, 50, Color.YELLOW);
```

Interfaces are another example of polymorphism. `a1` and `a2` are the same type of variable, but they refer to objects with different types. And similarly with `p1` and `p2`.

Classes may extend only one superclass, but they may implement as many interfaces as needed. Java library classes often implement multiple interfaces.

Event Listeners

Now that our `Drawing` is based on `Actor` instead of `DrawablePolygon`, we can draw other types of graphics. Here is the beginning of a class that reads an image from a file and shows the image moving across the canvas. The class is called `Sprite` because a moving image is sometimes called a **sprite**, in the context of computer graphics:

```
public class Sprite implements Actor, KeyListener {
    private int xpos;
    private int ypos;
    private int dx;
    private int dy;
    private Image image;

    public Sprite(String path, int xpos, int ypos) {
        this.xpos = xpos;
        this.ypos = ypos;
        try {
            this.image = ImageIO.read(new File(path));
        } catch (IOException exc) {
            exc.printStackTrace();
        }
    }
}
```

The instance variables `xpos` and `ypos` represent the location of the sprite. `dx` and `dy` represent the velocity of the sprite in the x and y directions.

The constructor takes as parameters the name of a file and the initial position. It uses `ImageIO`, from the `javax.imageio` package, to read the file. If an error occurs during

reading, an `IOException` is caught, and the program displays the stack trace for debugging.

`Sprite` implements two interfaces: `Actor` and `KeyListener`. `Actor` requires that we provide `draw` and `step` methods:

```
public void draw(Graphics g) {
    g.drawImage(image, xpos, ypos, null);
}

public void step() {
    xpos += dx;
    ypos += dy;
}
```

The `draw` method draws the image at the sprite's current position. The `step` method changes the position based on `dx` and `dy`, which are initially zero.

`KeyListener` is an interface for receiving keyboard events, which means we can detect and respond to key presses. A class that implements `KeyListener` has to provide the following methods:

```
void keyPressed(KeyEvent e)
```

Invoked when a key has been *pressed*. This method is invoked repeatedly while a key is being held down.

```
void keyReleased(KeyEvent e);
```

Invoked when a key has been *released*, meaning it is no longer down.

```
void keyTyped(KeyEvent e);
```

Invoked when a key has been *typed*, which generally means it has been both pressed and released.

These methods get invoked when the user presses and releases *any* key. They take a `KeyEvent` object as a parameter, which specifies which key was pressed, released, or typed.

We can use these methods to design a simple animation using the arrow keys. When the user presses up or down, the sprite will move up or down. When the user presses left or right, the sprite will move left or right.

Here's an implementation of `keyPressed` that uses a `switch` statement to test which arrow key was pressed and sets `dx` or `dy` accordingly. (There is no default branch, so we ignore all other keys.)

```
public void keyPressed(KeyEvent e) {
    switch (e.getKeyCode()) {
        case KeyEvent.VK_UP:
            dy = -5;
            break;
    }
}
```



```

        case KeyEvent.VK_DOWN:
            dy = +5;
            break;
        case KeyEvent.VK_LEFT:
            dx = -5;
            break;
        case KeyEvent.VK_RIGHT:
            dx = +5;
            break;
    }
}

```

The values of dx and dy determine how much the sprite moves each time step is invoked. While the user holds down an arrow key, the sprite will move at a constant speed.

Here's an implementation of `keyReleased` that runs when the user releases the key:

```

public void keyReleased(KeyEvent e) {
    switch (e.getKeyCode()) {
        case KeyEvent.VK_UP:
        case KeyEvent.VK_DOWN:
            dy = 0;
            break;
        case KeyEvent.VK_LEFT:
        case KeyEvent.VK_RIGHT:
            dx = 0;
            break;
    }
}

```

When the user releases the key, `keyReleased` sets dx or dy to 0, so the sprite stops moving in that direction.

We don't need the `keyTyped` method for this example, but it's required by the interface; if we don't provide one, the compiler will complain. So we provide an implementation that does nothing:

```

public void keyTyped(KeyEvent e) {
    // do nothing
}

```

Now, here's the code we need to create a `Sprite`, add it to a `Drawing`, and configure it as a `KeyListener`:

```

Sprite sprite = new Sprite("face-smile.png", 25, 150);
drawing.add(sprite);
drawing.addKeyListener(sprite);
drawing.setFocusable(true);

```

Recall that the `add` method is one that we wrote in “An Initial Drawing” on page 248. It adds an `Actor` to the list of objects to be drawn.

The `addKeyListener` method is inherited from `Canvas`. It adds a `KeyListener` to the list of objects that will receive key events.

In graphical applications, key events are sent to components only when they have the keyboard focus. The `setFocusable` method ensures that drawing will have the focus initially, without the user having to click it first.

Timers

Now that you know about interfaces and events, we can show you a better way to create animations. Previously, we implemented the animation loop by using `while (true)` and `Thread.sleep`. Java provides a `Timer` class (in `javax.swing`) that encapsulates this behavior.

A `Timer` is useful for executing code at regular intervals. The constructor for `Timer` takes two parameters:

- `int delay`
 // milliseconds between events
- `ActionListener listener`
 // for handling timer events

The `ActionListener` interface requires only one method, `actionPerformed`. This is the method the `Timer` invokes after the given delay.

Using a `Timer`, we can reorganize the code in `main` by defining a class that implements `ActionListener`:

```
public class VideoGame implements ActionListener {
    private Drawing drawing;

    public VideoGame() {
        Sprite sprite = new Sprite("face-smile.png", 50, 50);
        drawing = new Drawing(800, 600);
        drawing.add(sprite);
        drawing.addKeyListener(sprite);
        drawing.setFocusable(true);

        JFrame frame = new JFrame("Video Game");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(drawing);
        frame.pack();
        frame.setVisible(true);
    }
}
```

```

    public void actionPerformed(ActionEvent e) {
        drawing.step();
    }

    public static void main(String[] args) {
        VideoGame game = new VideoGame();
        Timer timer = new Timer(33, game);
        timer.start();
    }
}

```

The `main` method constructs a `VideoGame` object, which creates a `Sprite`, a `Drawing`, and a `JFrame`. Then it constructs a `Timer` object and starts the timer. Every 33 milliseconds, the `Timer` invokes `actionPerformed`, which invokes `step` on the `Drawing`.

`Drawing.step` invokes `step` on all of its `Actor` objects, which causes them to update their position, color, or other aspects of their appearance. The `Drawing.step` then repaints the `Canvas`, and the time step is done.

At this point, you have all of the elements you need to write your own video games. In the exercises at the end of this chapter, we have some suggestions for getting started.

We hope this final chapter has been a helpful summary of topics presented throughout the book, including input and output, decisions and loops, classes and methods, arrays and objects, inheritance, and graphics. Congratulations on making it to the end!

Vocabulary

generalization

The process of extracting common code from two or more classes and moving it into a superclass.

specialization

Extending a class to add new attributes or methods, or to modify existing behavior.

polymorphism

A language feature that allows objects to be assigned to variables of related types.

sprite

A computer graphic that may be moved or otherwise manipulated on the screen.

Exercises

The code for this chapter is in the `ch16` directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

The following exercises give you a chance to practice using the features in this chapter by extending the example code.

Exercise 17-1.

The `Polygon` class does not provide a `toString` method; it inherits the default `toString` from `java.lang.Object`, which includes only the class’s name and memory location. Write a more useful `toString` method for `DrawablePolygon` that includes its (x, y) points.

Exercise 17-2.

Write a class `MovingPolygon` that extends `RegularPolygon` and implements `Actor`. It should have instance variables `posx` and `posy` that specify its position, and `dx` and `dy` that specify its velocity (and direction). During each time step, it should update its position. If it gets to the edge of the `Drawing`, it should reverse direction by changing the sign of `dx` or `dy`.

Exercise 17-3.

Modify the `VideoGame` class so it displays a `Sprite` and a `MovingPolygon` (from the previous exercise). Add code that detects collisions between `Actor` objects in the same `Drawing`, and invoke a method on both objects when they collide. *Hint:* You might want to add a method to the `Actor` interface, guaranteeing that all `Actor` objects know how to handle collisions.

Exercise 17-4.

Java provides other event listeners that you can implement to make your programs interactive. For example, the interfaces `MouseListener`, `MouseMotionListener`, and `MouseWheelListener` allow you to respond to mouse input. Use the `MouseListener` interface to implement an `Actor` that can respond to mouse clicks.

The steps for compiling, running, and debugging Java code depend on your development environment and operating system. We avoided putting these details in the main text, because they can be distracting. Instead, we provide this appendix with a brief introduction to DrJava—an **integrated development environment (IDE)** that is helpful for beginners—and other development tools, including Checkstyle for code quality and JUnit for testing.

Installing DrJava

The easiest way to start programming in Java is to use a website that compiles and runs Java code in the browser. Examples include [Repl.it](#), [Trinket](#), [JDoodle](#), and others.

If you are unable to install software on your computer (which is often the case in public schools and Internet cafés), you can use these online development environments for almost everything in this book.

But if you want to compile and run Java programs on your own computer, you will need the following:

- The **Java Development Kit (JDK)**, which includes the compiler, the **Java Virtual Machine (JVM)** that interprets the compiled byte code, and other tools such as Javadoc.
- A **text editor** such as Atom, Notepad++, or Sublime Text, and/or an IDE such as DrJava, Eclipse, jGrasp, or NetBeans.

The JDK we recommend is OpenJDK, an open source implementation of Java SE (Standard Edition). The IDE we recommend is DrJava, which is an open source development environment written in Java (see [Figure A-1](#)).

To install OpenJDK, visit [the AdoptOpenJDK website](#). Download and run the installer for your operating system.

To install DrJava, visit [the DrJava website](#) and download the **JAR** file. We recommend that you save it to your desktop or another convenient location. Simply double-click the JAR file to run DrJava. Refer to the [DrJava documentation](#) for more details.

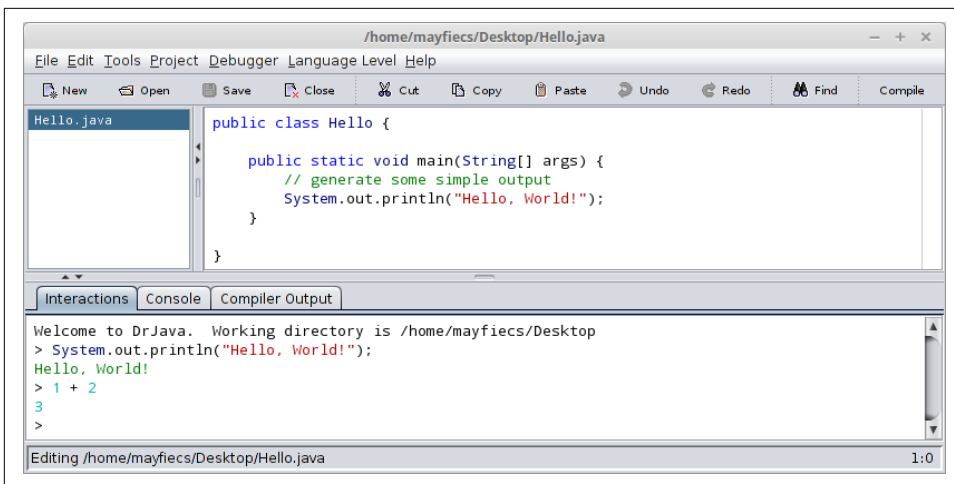


Figure A-1. DrJava editing the Hello World program

When running DrJava for the first time, we recommend you change three settings from the Edit > Preferences menu under Miscellaneous: set the Indent Level to 4, check the Automatically Close Block Comments box, and uncheck the Keep Emacs-style Backup Files box.

DrJava Interactions

One of the most useful features of DrJava is the *Interactions pane* at the bottom of the window. It provides the ability to try out code quickly, without having to write a class definition and save/compile/run the program. [Figure A-2](#) shows an example.

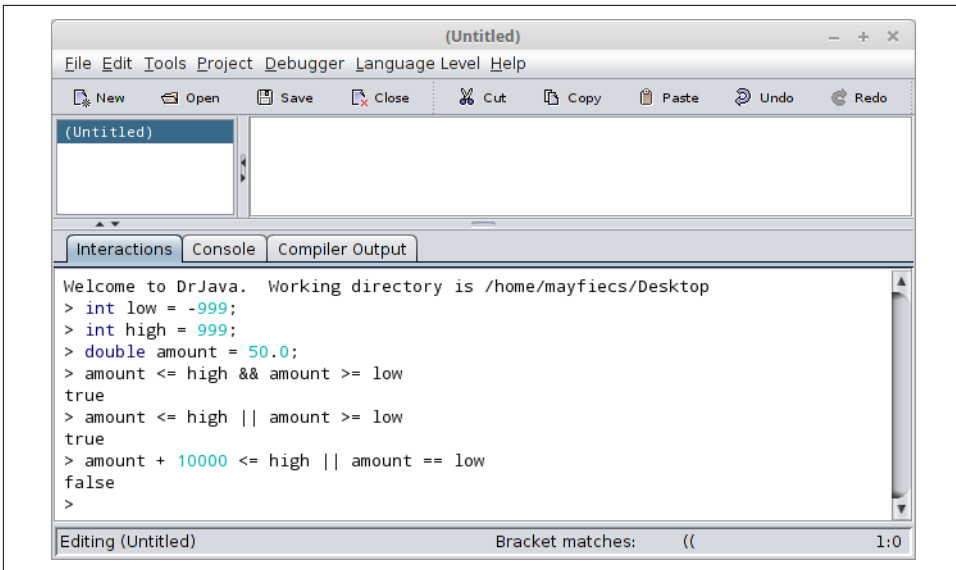


Figure A-2. The Interactions pane in DrJava

There is one subtle detail to note when using the Interactions pane. If you don't end an expression (or statement) with a semicolon, DrJava automatically displays its value. Notice in [Figure A-2](#) that the variable declarations end with semicolons, but the logic expressions in the following lines do not. This feature saves you from having to type `System.out.println` every time.

What's nice about this feature is that you don't have to create a new class, declare a `main` method, write arbitrary expressions inside `System.out.println` statements, save the source file, and get all of your code to compile in advance. Also, you can press the up/down arrows on the keyboard to repeat previous commands and experiment with incremental differences.

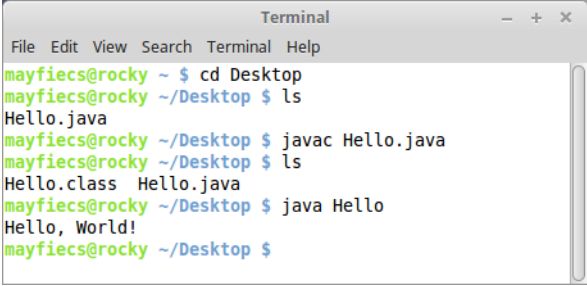
Command-Line Interface

One of the most powerful and useful skills you can learn is how to use the **command-line interface**, also called the *terminal*. The command line is a direct interface to the operating system. It allows you to run programs, manage files and directories, and monitor system resources. Many advanced tools, both for software development and general-purpose computing, are available only at the command line.

Many good tutorials are available online for learning the command line for your operating system; just search the web for “command line tutorial”. On Unix systems like Linux and macOS, you can get started with just four commands: change the

working directory (`cd`), list directory contents (`ls`), compile Java programs (`javac`), and run Java programs (`java`).

Figure A-3 shows an example in which the `Hello.java` source file is stored in the `Desktop` directory. After changing to that location and listing the files, we use the `javac` command to compile `Hello.java`. Running `ls` again, we see that the compiler generated a new file, `Hello.class`, which contains the byte code. We run the program by using the `java` command, which displays the output on the following line.



```
Terminal
File Edit View Search Terminal Help
mayfiecs@rocky ~ $ cd Desktop
mayfiecs@rocky ~/Desktop $ ls
Hello.java
mayfiecs@rocky ~/Desktop $ javac Hello.java
mayfiecs@rocky ~/Desktop $ ls
Hello.class Hello.java
mayfiecs@rocky ~/Desktop $ java Hello
Hello, World!
mayfiecs@rocky ~/Desktop $
```

Figure A-3. Compiling and running `Hello.java` from the command line

Note that the `javac` command requires a *filename* (or multiple source files separated by spaces), whereas the `java` command requires a single *class name*. If you use DrJava, it runs these commands for you behind the scenes and displays the output in the Interactions pane.

Taking time to learn this efficient and elegant way of interacting with the operating system will make you more productive. People who don't use the command line don't know what they're missing.

Command-Line Testing

As described in “[Debugging Programs](#)” on page 9, it's more effective to program and debug your code little by little than to attempt writing everything all at once. And after you've completed programming an algorithm, it's important to test that it works correctly on a variety of inputs.

Throughout the book, we illustrate techniques for testing your programs. Most, if not all, testing is based on a simple idea: does the program do what we expect it to do? For simple programs, it's not difficult to run them several times and see what happens. But at some point, you will get tired of typing the same test cases over and over.

We can automate the process of entering input and comparing *expected output* with *actual output* by using the command line. The basic idea is to store the test cases in

plain-text files and trick Java into thinking they are coming from the keyboard. Here are step-by-step instructions:

1. Make sure you can compile and run the *Convert.java* example in the *ch03* directory of *ThinkJavaCode2*. (See “Using the Code Examples” on page xii for instructions on how to download the repository.)
2. In the same directory as *Convert.java*, create a plain-text file named *test.in* (*in* is for *input*). Enter the following line and save the file:

```
193.04
```

3. Create a second plain-text file named *test.exp* (*exp* is for *expected*). Enter the following line and save the file:

```
193.04 cm = 6 ft, 4 in
```

4. Open a terminal, and change to the directory with these files. Run the following command to test the program:

```
java Convert < test.in > test.out
```

On the command line, `<` and `>` are **redirection operators**. The first one redirects the contents of *test.in* to *System.in*, as if it were entered from the keyboard. The second one redirects the contents of *System.out* to a new file *test.out*, much like a screen capture. In other words, the *test.out* file contains the output of your program.

By the way, it’s perfectly okay to compile your programs in DrJava (or another environment) and run them from the command line. Knowing both techniques allows you to use the right tool for the job.

At this point, we just need to compare the contents *test.out* with *test.exp*. If the files are the same, the program outputted what we expected it to output. If not, then we found a bug, and we can use the output to begin debugging our program. Fortunately, there’s a simple way to compare files on the command line:

```
diff test.exp test.out
```

The `diff` utility summarizes the differences between two files. If there are no differences, it displays nothing, which in our case is what we want. If the expected output differs from the actual output, we need to continue debugging. Usually, the program is at fault, and `diff` provides some insight about what is broken. But there’s also a chance that we have a correct program and the expected output is wrong.

Interpreting the results from `diff` can be confusing, but fortunately many graphical tools can show the differences between two files. For example, on Windows you can install `WinMerge`, on Mac you can use `opendiff` (which comes with Xcode), and on Linux there’s `meld`, shown in [Figure A-4](#).

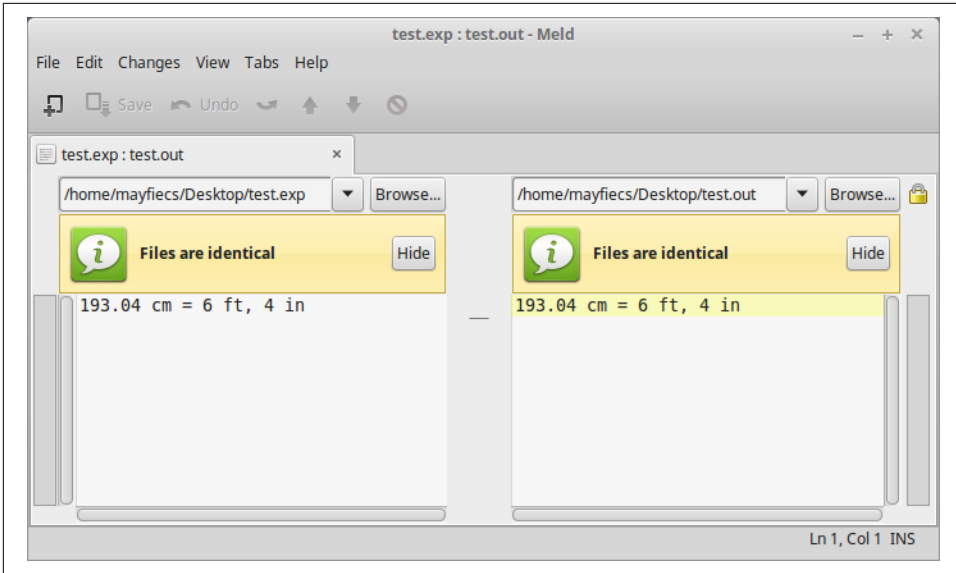


Figure A-4. Using `meld` to compare expected output with the actual output

Regardless of what tool you use, the goal is the same. Debug your program until the actual output is *identical* to the expected output.

Running Checkstyle

Checkstyle is a command-line tool that can be used to determine if your source code follows a set of style rules. It also checks for common programming mistakes, such as class and method design problems.

You can [download the latest version](#) as a JAR file. To run Checkstyle, move (or copy) the JAR file to the same directory as your program. Open a terminal in that location, and run the following command:

```
java -jar checkstyle-*-all.jar -c /google_checks.xml *.java
```

The `*` characters are **wildcards** that match whatever version of Checkstyle you have and whatever Java source files are present. The output indicates the file and line number of each problem. This example refers to a method beginning on line 93, column 5 of *Hello.java*:

```
Hello.java:93:5: Missing a Javadoc comment
```

The file `/google_checks.xml` is inside the JAR file and represents most of Google's style rules. You can alternatively use `/sun_checks.xml` or provide your own configuration file. See Checkstyle's website for more information.

If you apply Checkstyle to your source code often, you will likely internalize good style habits over time. But there are limits to what automatic style checkers can do. In particular, they can't evaluate the *quality* of your comments, the *meaning* of your variable names, or the *structure* of your algorithms.

Good comments make it easier for experienced developers to identify errors in your code. Good variable names communicate the intent of your program and how the data is organized. And good programs are designed to be efficient and demonstrably correct.

Tracing with a Debugger

A great way to visualize the flow of execution, including how parameters and arguments work, is to use a **debugger**. Most debuggers make it possible to do the following:

- Set a **breakpoint**, a line where you want the program to pause.
- Step through the code one line at a time and watch what it does.
- Check the values of variables and see when and how they change.

For example, open any program in DrJava and move the cursor to the first line of `main`. Press `Ctrl+B` to toggle a breakpoint on the current line; it should now be highlighted in red. Press `Ctrl+Shift+D` to turn on Debug mode; a new pane should appear at the bottom of the window. These commands are also available from the Debugger menu, in case you forget the shortcut keys.

When you run the program, execution pauses at the first breakpoint. The Debug pane displays the **call stack**, with the current method on top of the stack, as shown in [Figure A-5](#). You might be surprised to see how many methods were called before the `main` method!

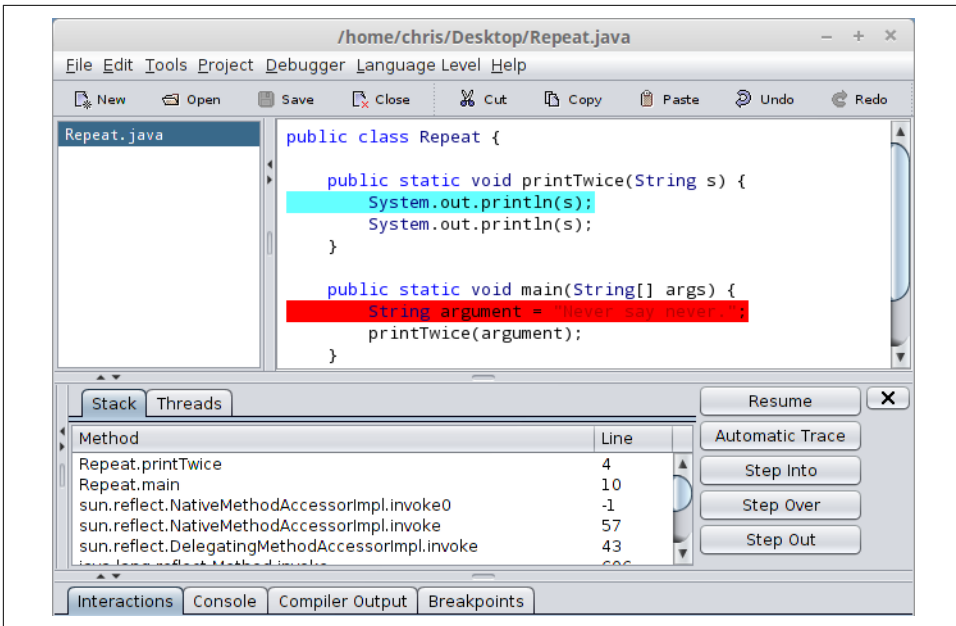


Figure A-5. The DrJava debugger. Execution is currently paused on the first line of `printTwice`. There is a breakpoint on the first line of `main`.

To the right are several buttons that allow you to step through the code at your own pace. You can also click `Automatic Trace` to watch DrJava run your code one line at a time.

Using a debugger is like having the computer proofread your code out loud. When the program is paused, you can examine (or even change) the value of any variable by using the `Interactions` pane.

Tracing allows you to follow the flow of execution and see how data passes from one method to another. You might expect the code do one thing, but then the debugger shows it doing something else. At that moment, you gain insight about what may be wrong with the code.

You can edit your code while debugging it, but we don't recommend it. If you add or delete multiple lines of code while the program is paused, the results can be confusing.

See the “[Debugger](#)” chapter of [DrJava's documentation](#) for more information about using the debugger feature of DrJava.

Testing with JUnit

When beginners start writing methods, they usually test them by invoking them from `main` and checking the results by hand. For example, to test `fibonacci` from “The Leap of Faith” on page 116, we could write this:

```
public static void main(String[] args) {
    if (fibonacci(1) != 1) {
        System.err.println("fibonacci(1) is incorrect");
    }
    if (fibonacci(2) != 1) {
        System.err.println("fibonacci(2) is incorrect");
    }
    if (fibonacci(3) != 2) {
        System.err.println("fibonacci(3) is incorrect");
    }
}
```

This test code is self-explanatory, but it’s longer than it needs to be, and it doesn’t scale very well. In addition, the error messages provide limited information. For cases where we know the right answer, we can do better by writing **unit tests**.

JUnit is a common testing tool for Java programs. To use it, you have to create a test class that contains test methods.

For example, suppose that the `fibonacci` method belongs to a class named `Series`. Here is the corresponding JUnit test class and test method:

```
import junit.framework.TestCase;

public class SeriesTest extends TestCase {

    public void testFibonacci() {
        assertEquals(1, Series.fibonacci(1));
        assertEquals(1, Series.fibonacci(2));
        assertEquals(2, Series.fibonacci(3));
    }
}
```

This example uses the keyword `extends`, which indicates that the new class, `SeriesTest`, is based on an existing class, `TestCase`. The `TestCase` class is imported from the package `junit.framework`.

The names in this example follow convention: if the name of your class is `Something`, the name of the test class should be `SomethingTest`. And if there is a method in `Something` named `someMethod`, there should be a method in `SomethingTest` named `testSomeMethod`.

Many development environments can generate test classes and test methods automatically. In DrJava, you can select New JUnit Test Case from the File menu to generate an empty test class.

`assertEquals` is provided by the `TestCase` class. It takes two arguments and checks whether they are equal. If so, it does nothing; otherwise, it displays a detailed error message. The first argument is the *expected value*, which we consider correct, and the second argument is the *actual value* we want to check. If they are not equal, the test fails.

Using `assertEquals` is more concise than writing your own `if` statements and `System.err` messages. JUnit provides additional assert methods, such as `assertNull`, `assertSame`, and `assertTrue`, which can be used to design a variety of tests.

To run JUnit directly from DrJava, click the Test button on the toolbar. If all your test methods pass, you will see a green bar in the lower-right corner. Otherwise, DrJava will take you directly to the first assertion that failed.

Vocabulary

IDE

An integrated development environment that includes tools for editing, compiling, and debugging programs.

JDK

The Java Development Kit, which contains the compiler, Javadoc, and other tools.

JVM

The Java Virtual Machine, which interprets the compiled byte code.

text editor

A program that edits plain-text files, the format used by most programming languages.

JAR

A Java Archive, which is essentially a ZIP file containing classes and other resources.

command-line interface

A means of interacting with the computer by issuing commands in the form of successive lines of text.

redirection operator

A command-line feature that substitutes *System.in* and/or *System.out* with a plain-text file.

wildcard

A command-line feature that allows you to specify a pattern of filenames by using the * character.

debugger

A tool that allows you to run one statement at a time and see the contents of variables.

breakpoint

A line of code at which the debugger will pause a running program.

call stack

The history of method calls and where to resume execution after each method returns.

unit test

Code that exercises a single method of a program, testing for correctness and/or efficiency.

Java programs have three types of comments:

End-of-line comments

These start with `//` and generally contain short phrases that explain specific lines of code.

Multiline comments

These start with `/*` and end with `*/`, and are typically used for copyright statements.

Documentation comments

These start with `/**` and end with `*/`, and describe what each class and method does.

End-of-line and multiline comments are written primarily for yourself. They help you remember specific details about your source code. Documentation comments, on the other hand, are written for others. They explain how to use your classes and methods in other programs.

A nice feature of the Java language is the ability to embed documentation in the source code itself. That way, you can write it as you go, and as things change, it is easier to keep the documentation consistent with the code.

You can extract documentation from your source code, and generate well-formatted HTML pages, using a tool called **Javadoc**. This tool is included with the Java compiler, and it is widely used. In fact, the [official documentation for the Java library](#) is generated by Javadoc.

Reading Documentation

As an example, let's look at the documentation for `Scanner`, a class we first used in “The `Scanner` Class” on page 30. You can find the documentation quickly by doing a web search for “Java `Scanner`”. Figure B-1 shows a screenshot of the page.

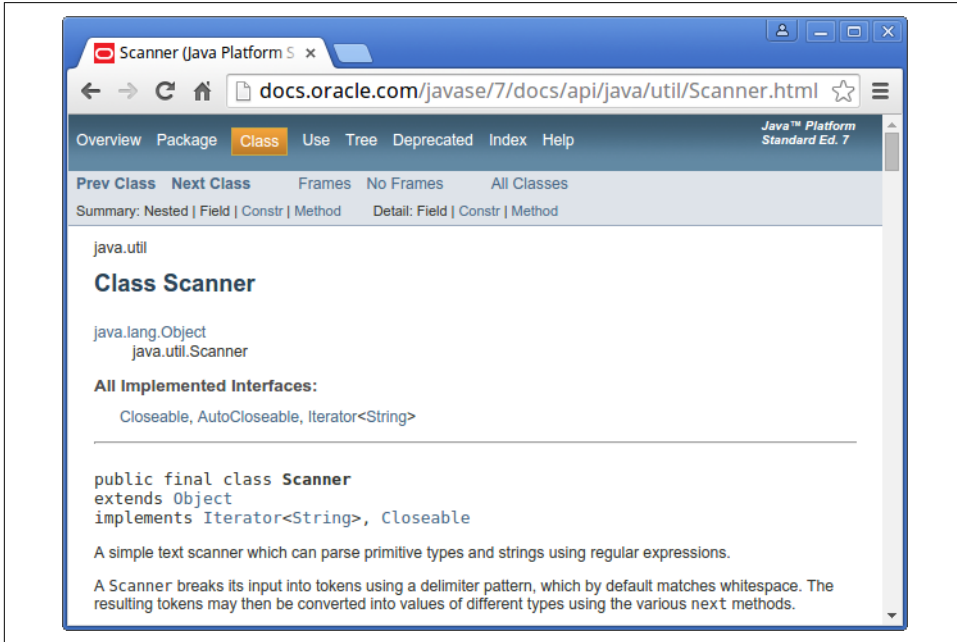


Figure B-1. The documentation for `Scanner`

Documentation for other classes uses a similar format. The first line is the package that contains the class, such as `java.util`. The second line is the name of the class. The All Implemented Interfaces section lists some of the functionality a `Scanner` has.

The next section of the documentation is a narrative that explains the purpose of the class and includes examples of how to use it. This text can be difficult to read, because it may use terms you have not yet learned. But the examples are often very useful. A good way to get started with a new class is to paste the examples into a test file and see if you can compile and run them.

One of the examples shows how you can use a `Scanner` to read input from a `String` instead of `System.in`:

```
String input = "1 fish 2 fish red fish blue fish";  
Scanner s = new Scanner(input);
```

After the narrative, code examples, and other details, you will find the following tables:

Constructor summary

Ways of creating, or constructing, a Scanner

Method summary

The list of methods that the Scanner class provides

Constructor detail

More information about how to create a Scanner

Method detail

More information about each method

For example, here is the summary information for `nextInt()`:

```
public int nextInt()  
    Scans the next token of the input as an int.
```

The first line is the method's **signature**, which specifies the name of the method, its parameters (none), and the type it returns (`int`). The next line is a short description of what it does.

The "Method detail" explains more:

```
public int nextInt()  
    Scans the next token of the input as an int.
```

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:
the `int` scanned from the input

Throws:
`InputMismatchException` - if the next token does not match the `Integer` regular expression, or is out of range
`NoSuchElementException` - if input is exhausted
`IllegalStateException` - if this scanner is closed

The Returns section describes the result when the method succeeds. In contrast, the Throws section describes possible errors and exceptions that may occur. Exceptions are said to be *thrown*, like a referee throwing a flag, or like a toddler throwing a fit.

It might take you some time to get comfortable reading documentation and learning which parts to ignore. But it's worth the effort. Knowing what's available in the library helps you avoid reinventing the wheel. And a little bit of documentation can save you a lot of debugging.

Writing Documentation

As you benefit from reading good documentation, you should “pay it forward” by writing good documentation.

Javadoc scans your source files looking for documentation comments, also known as *Javadoc comments*. They begin with `/**` (two stars) and end with `*/` (one star). Anything in between is considered part of the documentation.

Here’s a class definition with two Javadoc comments, one for the `Goodbye` class and one for the `main` method:

```
/**
 * Example program that demonstrates print vs println.
 */
public class Goodbye {

    /**
     * Prints a greeting.
     */
    public static void main(String[] args) {
        System.out.print("Goodbye, "); // note the space
        System.out.println("cruel world");
    }
}
```

The class comment explains the purpose of the class. The method comment explains what the method does.

Notice that this example also has an end-of-line comment (`//`). In general, these comments are short phrases that help explain complex parts of a program. They are intended for other programmers reading and maintaining the source code.

In contrast, Javadoc comments are longer, usually complete sentences. They explain what each method does, but they omit details about how the method works. And they are intended for people who will use the methods without looking at the source code.

Appropriate comments and documentation are essential for making source code readable. And remember that the person most likely to read your code in the future, and appreciate good documentation, is you.

Javadoc Tags

It’s generally a good idea to document each class and method, so that other programmers can understand what they do without having to read the code.

To organize the documentation into sections, Javadoc supports optional **tags** that begin with the at sign (`@`). For example, we can use `@author` and `@version` to provide information about the class:

```

/**
 * Utility class for extracting digits from integers.
 *
 * @author Chris Mayfield
 * @version 1.0
 */
public class DigitUtil {

```

Documentation comments should begin with a **description** of the class or method, followed by the tags. These two sections are separated by a blank line (not counting the *).

For methods, we can use `@param` and `@return` to provide information about parameters and return values:

```

/**
 * Tests whether x is a single digit integer.
 *
 * @param x the integer to test
 * @return true if x has one digit, false otherwise
 */
public static boolean isSingleDigit(int x) {

```

Figure B-2 shows part of the resulting HTML page generated by Javadoc. Notice the relationship between the Javadoc comment (in the source code) and the resulting documentation (in the HTML page).

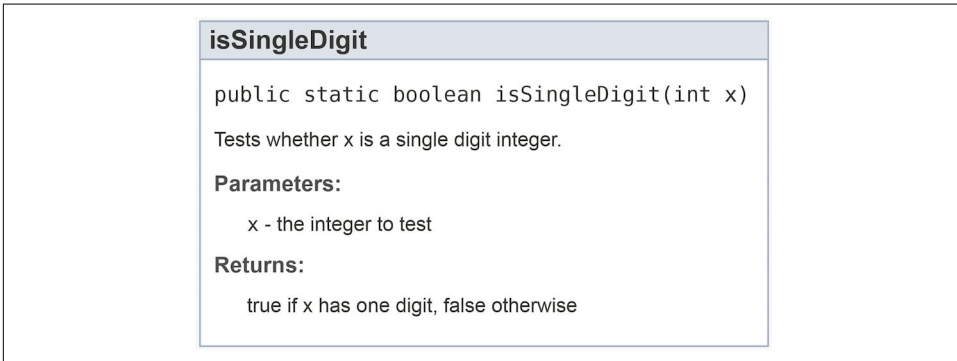


Figure B-2. HTML documentation for `isSingleDigit`

When writing parameter comments, do not include a hyphen (-) after the `@param` tag. Otherwise, you will have two hyphens in the resulting HTML documentation.

Notice also that the `@return` tag should not specify the type of the method. Comments like `@return boolean` are not useful, because you already know the return type from the method's signature.

Methods with multiple parameters should have separate `@param` tags that describe each one. Void methods should have no `@return` tag, since they do not return a value. Each tag should be on its own line in the source code.

Example Source File

Now let's take a look at a more complete example. The code for this section is in the `appb` directory of *ThinkJavaCode2*. See “Using the Code Examples” on page xii for instructions on how to download the repository.

Professional-grade source files often begin with a copyright statement. This text spans multiple lines, but it is not part of the documentation. So we use a multiline comment (`/*`) rather than a documentation comment (`/**`). Our example source file, `Convert.java`, includes the [MIT License](#):

```
/*
 * Copyright (c) 2019 Allen Downey and Chris Mayfield
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */
```

Import statements generally follow the copyright text. After that, we can define the class itself and begin writing the documentation (`/**`):

```
import java.util.Scanner;

/**
 * Methods for converting to/from the metric system.
 *
 * @author Allen Downey
 * @author Chris Mayfield
 * @version 6.1.5
 */
public class Convert {
```

A common mistake that beginners make is to put `import` statements between the documentation and the `public class` line. Doing so separates the documentation from the class itself. To avoid this issue, always make the end of the comment (the `*/`) “touch” the word `public`.

This class has two constants and three methods. The constants are self-explanatory, so there is no need to write documentation for them:

```
public static final double CM_PER_INCH = 2.54;

public static final int IN_PER FOOT = 12;
```

The methods, on the other hand, could use some explanation. Each documentation comment includes a description, followed by a blank line, followed by a `@param` tag for each parameter, followed by a `@return` tag:

```
/**
 * Converts a measurement in centimeters to inches.
 *
 * @param cm length in centimeters
 * @return length in inches
 */
public static double toImperial(double cm) {
    return cm / CM_PER_INCH;
}

/**
 * Converts a length in feet and inches to centimeters.
 *
 * @param feet how many feet
 * @param inches how many inches
 * @return length in centimeters
 */
public static double toMetric(int feet, int inches) {
    int total = feet * IN_PER FOOT + inches;
    return total * CM_PER_INCH;
}
```

The `main` method has a similar documentation comment, except there is no `@return` tag since the method is `void`:

```
/**
 * Tests the conversion methods.
 *
 * @param args command-line arguments
 */
public static void main(String[] args) {
    double cm, result;
    int feet, inches;
    Scanner in = new Scanner(System.in);
```

```

// test the Imperial conversion
System.out.print("Exactly how many cm? ");
cm = in.nextDouble();
result = toImperial(cm);
System.out.printf("That's %.2f inches\n", result);
System.out.println();

// test the Metric conversion
System.out.print("Now how many feet? ");
feet = in.nextInt();
System.out.print("And how many inches? ");
inches = in.nextInt();
result = toMetric(feet, inches);
System.out.printf("That's %.2f cm\n", result);
}

```

Here are two ways you can run the Javadoc tool on this example program:

- From the command line, go to the location for *Convert.java*. The `-d` option of `javadoc` indicates where to generate the HTML files:

```
javadoc -d doc Convert.java
```

- From DrJava, click the Javadoc button on the toolbar. The IDE will then prompt you for a location to generate the HTML files.

For more examples of what you can do with Javadoc comments, see the source code of any Java library class (e.g., `Scanner.java`). “[Java Library Source](#)” on page 152 explains how to find the source files for the Java library on your computer.

Vocabulary

documentation

Comments that describe the technical operation of a class or method.

Javadoc

A tool that reads Java source code and generates documentation in HTML format.

signature

The first line of a method that defines its name, return type, and parameters.

tag

A label that begins with an at sign (@) and is used by Javadoc to organize documentation into sections.

description

The first line of a documentation comment that explains what the class/method does.

The Java library includes the package `java.awt` for drawing 2D graphics. AWT stands for *Abstract Window Toolkit*. We are only going to scratch the surface of graphics programming. You can read more about it in the [Java tutorials](#).

Creating Graphics

There are several ways to create graphics in Java; the simplest way is to use `java.awt.Canvas` and `java.awt.Graphics`. A `Canvas` is a blank rectangular area of the screen onto which the application can draw. The `Graphics` class provides basic drawing methods such as `drawLine`, `drawRect`, and `drawString`.

Here is an example program that draws a circle by using the `fillOval` method:

```
import java.awt.Canvas;
import java.awt.Graphics;
import javax.swing.JFrame;

public class Drawing extends Canvas {

    public static void main(String[] args) {
        JFrame frame = new JFrame("My Drawing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Drawing drawing = new Drawing();
        drawing.setSize(400, 400);
        frame.add(drawing);
        frame.pack();
        frame.setVisible(true);
    }

    public void paint(Graphics g) {
```

```
        g.fillOval(100, 100, 200, 200);
    }
}
```

The `Drawing` class extends `Canvas`, so it has all the methods provided by `Canvas`, including `setSize`. You can read about the other methods in the documentation, which you can find by doing a web search for “Java Canvas”.

In the `main` method, we do the following:

1. Create a `JFrame` object, which is the window that will contain the canvas.
2. Create a `Drawing` object (which is the canvas), set its width and height, and add it to the frame.
3. Pack the frame (resize it) to fit the canvas, and display it on the screen.

Once the frame is visible, the `paint` method is called whenever the canvas needs to be drawn; for example, when the window is moved or resized. If you run this code, you should see a black circle on a gray background.

The application doesn't end after the `main` method returns; instead, it waits for the `JFrame` to close. When the `JFrame` closes, it calls `System.exit`, which ends the program.

Graphics Methods

You are probably used to Cartesian **coordinates**, where x and y values can be positive or negative. In contrast, Java uses a coordinate system where the origin is in the upper-left corner. That way, x and y can always be positive integers. [Figure C-1](#) shows these coordinate systems side by side.

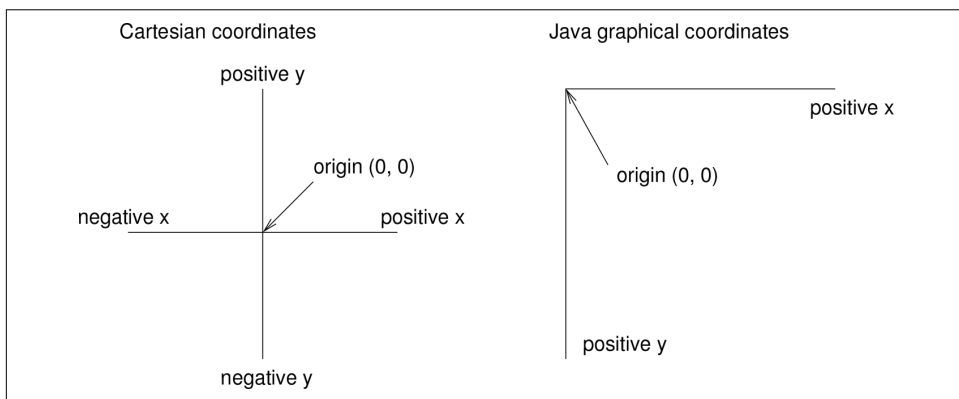


Figure C-1. The difference between Cartesian and Java graphical coordinates

Graphical coordinates are measured in **pixels**; each pixel corresponds to a dot on the screen.

To draw on the canvas, you invoke methods on a `Graphics` object. You don't have to create the `Graphics` object; it gets created when you create the `Canvas`, and it gets passed as an argument to `paint`.

The previous example used `fillOval`, which has the following signature:

```
/**
 * Fills an oval bounded by the specified rectangle with
 * the current color.
 */
public void fillOval(int x, int y, int width, int height)
```

The four parameters specify a **bounding box**, which is the rectangle in which the oval is drawn. `x` and `y` specify the location of the upper-left corner of the bounding box. The bounding box itself is not drawn (see [Figure C-2](#)).

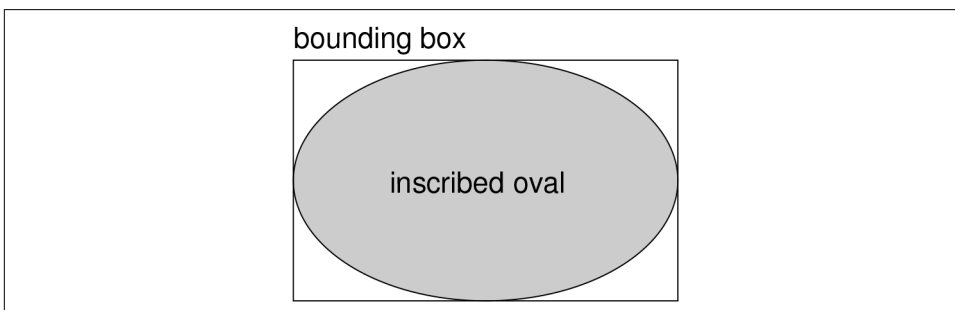


Figure C-2. An oval inside its bounding box

To choose the color of a shape, invoke `setColor` on the `Graphics` object:

```
g.setColor(Color.RED);
```

The `setColor` method determines the color of everything that gets drawn afterward. `Color.red` is a constant provided by the `Color` class; to use it, you have to `import java.awt.Color`. Other colors include the following:

BLACK	BLUE	CYAN	DARKGRAY	GRAY	LIGHTGRAY
GREEN	MAGENTA	ORANGE	PINK	WHITE	YELLOW

You can create your own colors by specifying the red, green, and blue (**RGB**) components. For example:

```
Color purple = new Color(128, 0, 128);
```

Each value is an integer in the range 0 (darkest) to 255 (lightest). The color (0, 0, 0) is black, and (255, 255, 255) is white.

You can set the background color of the Canvas by invoking `setBackground`:

```
canvas.setBackground(Color.WHITE);
```

Example Drawing

Suppose we want to draw a **Hidden Mickey**, which is an icon that represents Mickey Mouse. We can use the oval we just drew as the face, and then add two ears. To make the code more readable, let's use `Rectangle` objects to represent bounding boxes.

Here's a method that takes a `Rectangle` and invokes `fillOval`:

```
public void boxOval(Graphics g, Rectangle bb) {  
    g.fillOval(bb.x, bb.y, bb.width, bb.height);  
}
```

And here's a method that draws Mickey Mouse:

```
public void mickey(Graphics g, Rectangle bb) {  
    boxOval(g, bb);  
  
    int hx = bb.width / 2;  
    int hy = bb.height / 2;  
    Rectangle half = new Rectangle(bb.x, bb.y, hx, hy);  
  
    half.translate(-hx / 2, -hy / 2);  
    boxOval(g, half);  
  
    half.translate(hx * 2, 0);  
    boxOval(g, half);  
}
```

The first line draws the face. The next three lines create a smaller rectangle for the ears. We `translate` the rectangle up and left for the first ear, then to the right for the second ear. The result is shown in **Figure C-3**.

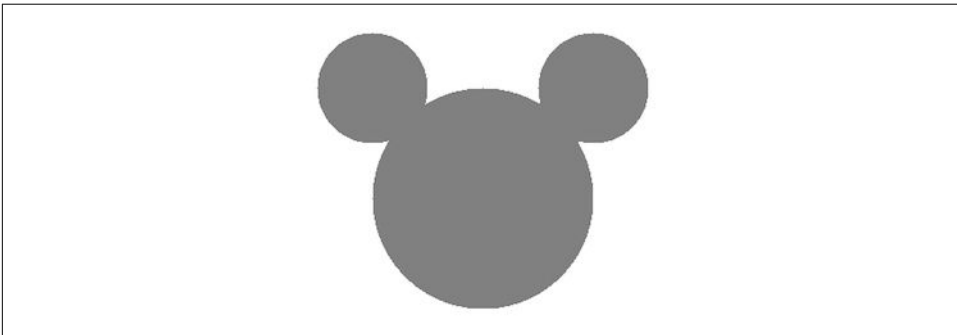


Figure C-3. A Hidden Mickey drawn using Java graphics

You can read more about `Rectangle` and `translate` in [Chapter 10](#). See the exercises at the end of this appendix for more example drawings.

Vocabulary

AWT

The Abstract Window Toolkit, a Java package for creating graphical user interfaces.

coordinate

A value that specifies a location in a 2D graphical window.

pixel

The unit in which coordinates are measured.

bounding box

A way to specify the coordinates of a rectangular area.

RGB

A color model based on adding red, green, and blue light.

Exercises

The code for this chapter is in the *appc* directory of *ThinkJavaCode2*. See “[Using the Code Examples](#)” on [page xii](#) for instructions on downloading the repository. Before you start the exercises, we recommend that you compile and run the examples.

Exercise C-1.

Draw the flag of Japan: a red circle on a white background that is wider than it is tall.

Exercise C-2.

Modify *Mickey.java* to draw ears on the ears, and ears on those ears, and more ears all the way down until the smallest ears are only 3 pixels wide. The result should look like Mickey Moose, shown in [Figure C-4](#).

Hint: You should have to add or modify only a few lines of code.

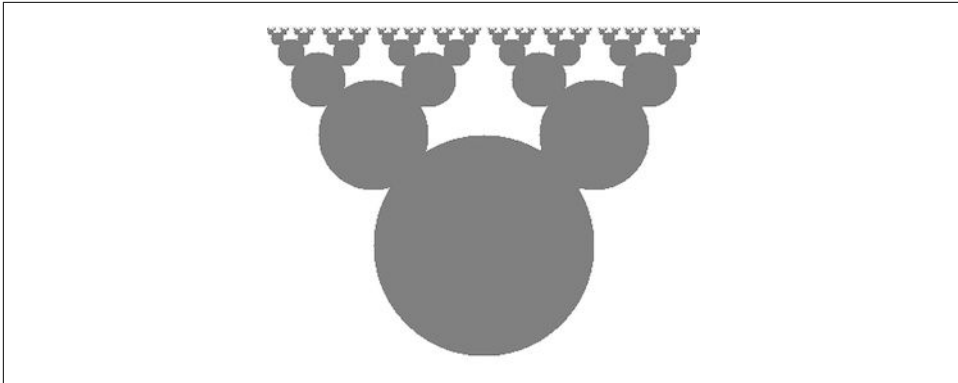


Figure C-4. A recursive shape we call Mickey Moose

Exercise C-3.

In this exercise, you will draw Moiré patterns that seem to shift around as you move. For an explanation of what is going on, see [Wikipedia's "Moiré pattern" entry](#).

1. Open `Moire.java` and read the `paint` method. Draw a sketch of what you expect it to do. Now run it. Did you get what you expected?
2. Modify the program so that the space between the circles is larger or smaller. See what happens to the image.
3. Modify the program so that the circles are drawn in the center of the screen and concentric, as in [Figure C-5](#) (left). The distance between the circles should be small enough that the Moiré interference is apparent.

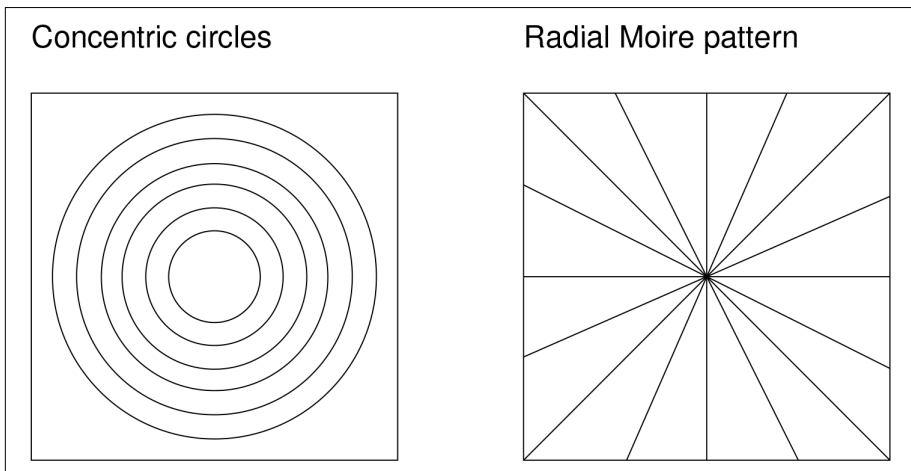


Figure C-5. Graphical patterns that can exhibit Moiré interference

4. Write a method named `radial` that draws a radial set of line segments as shown in [Figure C-5](#) (right), but they should be close enough together to create a Moiré pattern.
5. Just about any kind of graphical pattern can generate Moiré-like interference patterns. Play around and see what you can create.

Debugging

Although there are debugging suggestions throughout the book, we thought it would be useful to say more in an appendix. If you are having a hard time debugging, you might want to review this appendix from time to time.

The best debugging strategy depends on what kind of error you have:

- **Compile-time errors** indicate that there is something wrong with the syntax of the program. Example: omitting the semicolon at the end of a statement.
- **Run-time errors** are produced if something goes wrong while the program is running. Example: an infinite recursion eventually causes a `StackOverflowError`.
- **Logic errors** cause the program to do the wrong thing. Example: an expression may not be evaluated in the order you expect.

The following sections are organized by error type; some techniques are useful for more than one type.

Compile-Time Errors

The best kind of debugging is the kind you don't have to do because you avoid making errors in the first place. Incremental development, which we presented in “[Incremental Development](#)” on page 54, can help. The key is to start with a working program and add small amounts of code at a time. When there is an error, you will have a pretty good idea of where it is.

Nevertheless, you might find yourself in one of the following situations. For each situation, we have some suggestions about how to proceed.

The compiler is spewing error messages.

If the compiler reports 100 error messages, that doesn't mean there are 100 errors in your program. When the compiler encounters an error, it often gets thrown off track for a while. It tries to recover and pick up again after the first error, but sometimes it reports spurious errors.

Only the first error message is truly reliable. We suggest that you fix only one error at a time and then recompile the program. You may find that one semicolon or brace “fixes” 100 errors.

I'm getting a weird compiler message, and it won't go away.

First of all, read the error message carefully. It may be written in terse jargon, but often there is a carefully hidden kernel of information.

If nothing else, the message will tell you where in the program the problem occurred. Actually, it tells you where the compiler was when it noticed a problem, which is not necessarily where the error is. Use the information the compiler gives you as a guideline, but if you don't see an error where the compiler is pointing, broaden the search.

Generally, the error will be prior to the location of the error message, but in some cases it will be somewhere else entirely. For example, if you get an error message at a method invocation, the actual error may be in the method definition itself.

If you don't find the error quickly, take a breath and look more broadly at the entire program. Make sure the program is indented properly; that makes it easier to spot syntax errors.

Now, start looking for common syntax errors:

1. Check that all parentheses and brackets are balanced and properly nested. All method definitions should be nested within a class definition. All program statements should be within a method definition.
2. Remember that uppercase letters are not the same as lowercase letters.
3. Check for semicolons at the end of statements (and no semicolons after curly braces).
4. Make sure that any strings in the code have matching quotation marks. Make sure that you use double quotes for strings, and single quotes for characters.
5. For each assignment statement, make sure that the type on the left is the same as the type on the right. Make sure that the expression on the left is a variable name or something else that you can assign a value to (like an element of an array).

6. For each method invocation, make sure that the arguments you provide are in the right order and have the right type, and that the object you are invoking the method on is the right type.
7. If you are invoking a value method, make sure you are doing something with the result. If you are invoking a void method, make sure you are *not* trying to do something with the result.
8. If you are invoking an instance method, make sure you are invoking it on an object with the right type. If you are invoking a static method from outside the class where it is defined, make sure you specify the class name (using dot notation).
9. Inside an instance method, you can refer to the instance variables without specifying an object. If you try that in a static method—with or without `this`—you get a message like `nonstatic variable x cannot be referenced from a static context`.

If nothing works, move on to the next section....

I can't get my program to compile no matter what I do.

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your development environment to make sure the program you are editing is the program the compiler is compiling.

This situation is often the result of having multiple copies of the same program. You might be editing one version of the file but compiling a different version.

If you are not sure, try putting an obvious and deliberate syntax error right at the beginning of the program. Now compile again. If the compiler doesn't find the new error, there is probably something wrong with the way you set up the development environment.

If you have examined the code thoroughly, and you are sure the compiler is compiling the right source file, it is time for desperate measures—**Debugging by bisection**:

1. Make a backup of the file you are working on. If you are working on `Bob.java`, make a copy called `Bob.java.old`.
2. Delete about half the code from `Bob.java`. Try compiling again.
 - If the program compiles now, you know the error is in the code you deleted. Bring back about half of what you deleted and repeat.

- If the program still doesn't compile, the error must be in the code that remains. Delete about half of the remaining code and repeat.
3. Once you have found and fixed the error, start bringing back the code you deleted, a little bit at a time.

This process is ugly, but it goes faster than you might think and is very reliable. It works for other programming languages too!

I did what the compiler told me to do, but it still doesn't work.

Some error messages come with tidbits of advice, like `class Golfer must be declared abstract`. It does not define `int compareTo(java.lang.Object)` from interface `java.lang.Comparable`. It sounds like the compiler is telling you to declare `Golfer` as an abstract class, and if you are reading this book, you probably don't know what that is or how to do it.

Fortunately, the compiler is wrong. The solution in this case is to make sure `Golfer` has a method called `compareTo` that takes an `Object` as a parameter.

Don't let the compiler lead you by the nose. Error messages give you evidence that something is wrong, but the remedies they suggest are unreliable.

Run-Time Errors

It's not always clear what causes a run-time error, but you can often figure things out by adding print statements to your program.

My program hangs.

If a program stops and seems to be doing nothing, we say it is *hanging*. Often that means it is caught in an infinite loop or an infinite recursion.

- If you suspect that a particular loop is the problem, add a print statement immediately before the loop that says `entering the loop` and another immediately after that says `exiting the loop`.

Run the program. If you get the first message and not the second, you know where the program is getting stuck. Go to the [“Infinite loop” on page 291](#) section.

- Most of the time, an infinite recursion will cause the program to run for a while and then produce a `StackOverflowError`. If that happens, go to the [“Infinite recursion” on page 291](#) section.

If you are not getting a `StackOverflowError`, but you suspect there is a problem with a recursive method, you can still use the techniques in the “Infinite recursion” section.

- If neither of the previous suggestions helps, you might not understand the flow of execution in your program. Go to [“Flow of execution” on page 292](#).

Infinite loop

If you think you have an infinite loop and you know which loop it is, add a print statement at the end of the loop that displays the values of the variables in the condition, and the value of the condition.

For example:

```
while (x > 0 && y < 0) {
    // do something to x
    // do something to y

    System.out.println("x: " + x);
    System.out.println("y: " + y);
    System.out.println("condition: " + (x > 0 && y < 0));
}
```

Now when you run the program, you see three lines of output for each time through the loop. The last time through the loop, the condition should be `false`. If the loop keeps going, you will see the values of `x` and `y`, and you might figure out why they are not getting updated correctly.

Infinite recursion

Most of the time, an infinite recursion will cause the program to throw a `StackOverflowError`. But if the program is slow, it may take a long time to fill the stack.

If you know which method is causing an infinite recursion, check that there is a base case. There should be a condition that makes the method return without making a recursive invocation. If not, you need to rethink the algorithm and identify a base case.

If there is a base case, but the program doesn't seem to be reaching it, add a print statement at the beginning of the method that displays the parameters. Now when you run the program, you see a few lines of output every time the method is invoked, and you can see the values of the parameters. If the parameters are not moving toward the base case, you might see why not.

Flow of execution

If you are not sure how the flow of execution is moving through your program, add print statements to the beginning of each method with a message like `entering method foo`, where `foo` is the name of the method. Now when you run the program, it displays a trace of each method as it is invoked.

You can also display the arguments each method receives. When you run the program, check whether the values are reasonable, and check for one of the most common errors—providing arguments in the wrong order.

When I run the program, I get an exception.

When an exception occurs, Java displays a message that includes the name of the exception, the line of the program where the exception occurred, and a *stack trace*. The stack trace includes the method that was running, the method that invoked it, the method that invoked that one, and so on.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened:

NullPointerException

You tried to access an instance variable or invoke a method on an object that is currently `null`. You should figure out which variable is `null` and then figure out how it got to be that way.

Remember that when you declare a variable with an array type, its elements are initially `null` until you assign a value to them. For example, this code causes a `NullPointerException`:

```
int[] array = new Point[5];
System.out.println(array[0].x);
```

ArrayIndexOutOfBoundsException

The index you are using to access an array is either negative or greater than `array.length - 1`. If you can find the site where the problem is, add a print statement immediately before it to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

Now work your way backward through the program and see where the array and the index come from. Find the nearest assignment statement and see if it is doing the right thing. If either one is a parameter, go to the place where the method is invoked and see where the values are coming from.

StackOverflowError

See “[Infinite recursion](#)” on page 291.

FileNotFoundException

This means Java didn't find the file it was looking for. If you are using a project-based development environment like Eclipse, you might have to import the file into the project. Otherwise, make sure the file exists and that the path is correct. This problem depends on your filesystem, so it can be hard to track down.

ArithmeticException

Something went wrong during an arithmetic operation; for example, division by zero.

I added so many print statements I get inundated with output.

One of the problems with using print statements for debugging is that you can end up buried in output. There are two ways to proceed: either simplify the output or simplify the program.

To simplify the output, you can remove or comment out print statements that aren't helping, or combine them, or format the output so it is easier to understand. As you develop a program, you should write code to generate concise, informative traces of what the program is doing.

To simplify the program, scale down the problem the program is working on. For example, if you are sorting an array, sort a *small* array. If the program takes input from the user, give it the simplest input that causes the error.

Also, clean up the code. Remove unnecessary or experimental parts, and reorganize the program to make it easier to read. For example, if you suspect that the error is in a deeply nested part of the program, rewrite that part with a simpler structure. If you suspect a large method, split it into smaller methods and test them separately.

The process of finding the minimal test case often leads you to the bug. For example, if you find that a program works when the array has an even number of elements, but not when it has an odd number, that gives you a clue about what is going on.

Reorganizing the program can help you find subtle bugs. If you make a change that you think doesn't affect the program, and it does, that can tip you off.

Logic Errors

My program doesn't work.

Logic errors are hard to find because the compiler and interpreter provide no information about what is wrong. Only you know what the program is supposed to do, and only you know that it isn't doing it.

The first step is to make a connection between the code and the behavior you get. You need a hypothesis about what the program is actually doing. Here are some questions to ask yourself:

- Is there something the program was supposed to do that doesn't seem to be happening? Find the section of the code that performs that function, and make sure it is executing when you think it should. See [“Flow of execution” on page 292](#).
- Is something happening that shouldn't? Find code in your program that performs that function, and see if it is executing when it shouldn't.
- Is a section of code producing an unexpected effect? Make sure you understand the code, especially if it invokes methods in the Java library. Read the documentation for those methods, and try them out with simple test cases. They might not do what you think they do.

To program, you need a mental model of what your code does. If it doesn't do what you expect, the problem might not be the program; it might be in your head.

The best way to correct your mental model is to break the program into components (usually the classes and methods) and test them independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Here are some common logic errors to check for:

- Remember that integer division always rounds toward zero. If you want fractions, use `double`. More generally, use integers for countable things and floating-point numbers for measurable things.
- Floating-point numbers are only approximate, so don't rely on them to be perfectly accurate. You should probably never use the `==` operator with doubles. Instead of writing `if (d == 1.23)`, do something like `if (Math.abs(d - 1.23) < .000001)`.
- When you apply the equality operator (`==`) to objects, it checks whether they are identical. If you meant to check equivalence, you should use the `equals` method instead.
- By default for user-defined types, `equals` checks identity. If you want a different notion of equivalence, you have to override it.
- Inheritance can lead to subtle logic errors, because you can run inherited code without realizing it. See [“Flow of execution” on page 292](#).

I've got a big, hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables:

```
rect.setLocation(rect.getLocation().translate(
    -rect.getWidth(), -rect.getHeight()));
```

This example can be rewritten as follows:

```
int dx = -rect.getWidth();
int dy = -rect.getHeight();
Point location = rect.getLocation();
Point newLocation = location.translate(dx, dy);
rect.setLocation(newLocation);
```

The second version is easier to read, partly because the variable names provide additional documentation. It's also easier to debug, because you can check the types of the temporary variables and display their values.

Another problem that can occur with big expressions is that the order of operations may not be what you expect. For example, to evaluate $\frac{x}{2\pi}$, you might write this:

```
double y = x / 2 * Math.PI;
```

That is not correct, because multiplication and division have the same precedence, and they are evaluated from left to right. This code computes $\frac{x}{2}\pi$.

If you are not sure of the order of operations, check the documentation, or use parentheses to make it explicit:

```
double y = x / (2 * Math.PI);
```

This version is correct, and more readable for other people who haven't memorized the order of operations.

My method doesn't return what I expect.

If you have a return statement with a complex expression, you don't have a chance to display the value before returning:

```
public Rectangle intersection(Rectangle a, Rectangle b) {
    return new Rectangle(
        Math.min(a.x, b.x), Math.min(a.y, b.y),
        Math.max(a.x + a.width, b.x + b.width)
            - Math.min(a.x, b.x)
        Math.max(a.y + a.height, b.y + b.height)
            - Math.min(a.y, b.y));
}
```

Instead of writing everything in one statement, use temporary variables:

```

public Rectangle intersection(Rectangle a, Rectangle b) {
    int x1 = Math.min(a.x, b.x);
    int y1 = Math.min(a.y, b.y);
    int x2 = Math.max(a.x + a.width, b.x + b.width);
    int y2 = Math.max(a.y + a.height, b.y + b.height);
    Rectangle rect = new Rectangle(x1, y1, x2 - x1, y2 - y1);
    return rect;
}

```

Now you have the opportunity to display any of the intermediate variables before returning. And by reusing `x1` and `y1`, you made the code smaller too.

My print statement isn't doing anything.

If you use the `println` method, the output is displayed immediately, but if you use `print` (at least in some environments), the output gets stored without being displayed until the next newline. If the program terminates without displaying a newline, you may never see the stored output. If you suspect that this is happening, change some or all of the `print` statements to `println`.

I'm really, really stuck and I need help.

First, get away from the computer for a few minutes. Computers emit waves that affect the brain, causing the following symptoms:

- Frustration and rage.
- Superstitious beliefs (“the computer hates me”) and magical thinking (“the program works only when I wear my hat backward”).
- Sour grapes (“this program is lame anyway”).

If you suffer from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. People often find bugs when they let their mind wander. Good places to find bugs are buses, showers, and bed.

No, I really need help.

It happens. Even the best programmers get stuck. Sometimes you need another pair of eyes. Before you bring someone else in, make sure you have tried the techniques described in this appendix.

Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have print statements in the appropri-

ate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, give them the information they need:

- What kind of bug is it? Compile-time, run-time, or logic?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the test case that fails?
- If the bug occurs at compile time or run-time, what is the error message, and what part of the program does it indicate?
- What have you tried, and what have you learned?

By the time you explain the problem to someone, you might see the answer. This phenomenon is so common that some people recommend a debugging technique called *rubber ducking*. Here's how it works:

1. Buy a standard-issue rubber duck.
2. When you are really stuck on a problem, put the rubber duck on the desk in front of you and say, "Rubber duck, I am stuck on a problem. Here's what's happening..."
3. Explain the problem to the rubber duck.
4. Discover the solution.
5. Thank the rubber duck.

We're not kidding, it works!

I found the bug!

When you find the bug, the way to fix it is usually obvious. But not always. Sometimes what seems to be a bug is really an indication that you don't understand the program, or your algorithm contains an error. In these cases, you might have to rethink the algorithm or adjust your mental model. Take some time away from the computer to think, work through test cases by hand, or draw diagrams to represent the computation.

After you fix the bug, don't just start in making new errors. Take a minute to think about what kind of bug it was, why you made the error, how the error manifested itself, and what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly. Or even better, you will learn to avoid that type of bug for good.

Symbols

% remainder operator, 37
() parentheses, 22
; semicolon, 3, 23
<> angle brackets, 198
= assignment operator, 16
== equals operator, 63, 168
[] square brackets, 96
{ } curly braces, 4, 65

A

abecedarian, 92
abstract, 239
abstract class, 241
accessor, 166
accumulator, 101, 107
addition
 integer, 19
 string, 22
 time, 170
address, 29, 40, 168
algorithm, 8, 12
alias, 99, 107
aliasing, 194
allocate, 96, 107
anagram, 110
and operator, 67
angle brackets, 198
Append.java, 156
args, 134
argument, 47, 52, 56
ArithmeticException, 24, 293
array, 96, 107
 2D, 186

 copying, 99
 element, 97
 index, 97
 length, 100
 of cards, 189
 of objects, 181
 of strings, 177
 printing, 98
ArrayIndexOutOfBoundsException, 98, 292
ArrayList, 197, 206
Arrays class, 98, 99
assignment, 16, 25, 63
attribute, 148, 157
automatic conversion, 20, 36, 48, 63
AWT, 147, 279, 283

B

base case, 113, 122
BigInteger, 136
binary, 118, 123
binary search, 183, 185
block, 64, 74
Boole, George, 63
boolean, 63, 70, 73
bottom-up design, 212, 216
bounding box, 281, 283
brackets
 angle, 198
 curly, 4
 square, 96
branch, 64, 74
breakpoint, 265, 269
bug, 9, 12
byte code, 5, 11

C

- call stack, 114, 265, 269
- camel case, 46
- Canvas, 279
- Card, 175
- CardCollection, 206
- case-sensitive, 3, 16, 46, 89
- catch, 227
- chaining, 65, 74
- char, 6, 84
- Character, 132, 135
- charAt, 84
- Checkstyle, 264
- class, 4, 10, 172
 - Canvas, 279
 - Card, 175
 - CardCollection, 206
 - Deck, 189
 - definition, 161
 - Eights, 213
 - Graphics, 279
 - Hand, 209
 - JFrame, 280
 - Math, 51
 - Pile, 198
 - Player, 211
 - Point, 147
 - Rectangle, 149
 - relationships, 215
 - Scanner, 30
 - System, 29
 - Time, 161
 - utility, 30
 - wrapper, 132
- class diagram, 153, 157, 196, 215
- class variable, 178, 185
- client, 165, 172
- CodingBat, 120
- collection, 197, 201
- Color, 281
- command-line interface, 134, 261, 268
- comment, 11, 73
 - documentation, 271, 274
 - end-of-line, 4, 271
 - multi-line, 271
- compareTo, 89, 180
- comparison operator, 63
- compile, 5, 11, 288
- compile-time error, 23, 26, 287

- complete ordering, 179
- composition, 52, 57, 215
- computer science, 8, 12
- concatenate, 22, 26, 141
- concrete class, 241
- conditional statement, 64, 73
- constant, 33, 41
- constructor, 162, 172, 176, 189, 193
 - value, 164
- Convert.java, 38, 276
- coordinate, 147, 280, 283
- countdown, 111
- counter, 103
- CPU, 1
- Crazy Eights, 205

D

- De Morgan's laws, 69, 74
- debugger, 265, 269
- debugging, 9, 12, 287
 - by bisection, 289
 - experimental, 9
 - rubber duck, 297
- Deck, 189
- declaration, 15, 25, 147
- decrement, 81, 90
- degrees, 51
- dependent, 165
- description, 275, 278
- design process, 54, 137, 141, 191, 212
- deterministic, 102, 107
- diagram
 - class, 153, 196, 215
 - memory, 18, 87, 99, 130, 147, 163, 169, 177, 189
 - stack, 50, 113, 115, 120
- divisible, 37
- division
 - integer, 20, 21
- documentation, 38, 153, 272, 278
 - Javadoc comments, 274
 - Javadoc tags, 274
- dot notation, 148, 157
- double, 20
- Double, 132
- doubloon, 92
- Doubloon.java, 106
- DrJava, 259

E

Echo.java, 30
efficiency, 104, 143, 183, 192, 201, 207
Eights, 213
element, 96, 97, 107
empty array, 134, 141
empty string, 86, 90
encapsulate, 138, 141
encapsulation, 144, 150
 and generalization, 137
encode, 176, 185
enhanced for loop, 104, 108
equals, 88, 168, 169
equivalent, 168, 172, 179
error
 compile-time, 23, 287
 logic, 24, 287, 294
 message, 9, 23, 23, 288
 rounding, 21
 run-time, 24, 287
 syntax, 288
escape sequence, 8, 11, 84
exception, 24, 287, 292
 Arithmetic, 24
 ArrayIndexOutOfBoundsException, 98
 InputMismatch, 72
 MissingFormatArgument, 35
 NegativeArraySize, 96
 NullPointerException, 131, 135, 181
 NumberFormatException, 133
 StackOverflow, 113
 StringIndexOutOfBoundsException, 86, 136
Exception
 Interrupted, 226
executable, 5, 11
experimental debugging, 9
expression, 19, 26, 52, 52, 69
 big and hairy, 295
extends, 208, 208
extract digits, 37

F

factorial, 114, 123, 142
fibonacci, 117
FileNotFoundException, 293
final, 33, 179, 180
flag, 70, 74
floating-point, 20, 26
flow of execution, 46, 56, 292

for, 81
 enhanced, 104
format specifier, 34, 41, 83
format string, 34, 41, 167
frame, 50, 57

G

garbage collection, 155, 157, 178
generalization, 144, 150, 243, 257
generalize, 138, 141
getter, 166, 172
GitHub, xii
Goodbye.java, 6
Google style, 7
Graphics, 223, 279
Greenfield, Larry, 9
GuessStarter.java, 42

H

hacker, 71, 74, 135
Hand, 209
hanging, 290
hardware, 1, 10
HAS-A, 215, 216, 223, 240
Hello.java, 3
helper method, 191, 200
hexadecimal, 29, 34, 168
high-level language, 4, 11
histogram, 103, 108, 186
HTML, 153, 271, 275

I

IDE, 259, 268
identical, 168, 172
if statement, 64
immutable, 131, 141, 155, 180
import statement, 30, 40
increment, 81, 90
incremental development, 54, 57
independent, 165
index, 90, 97, 107, 182
indexOf, 87
infinite loop, 80, 90, 290
infinite recursion, 163, 290, 291
information hiding, 162, 172
inheritance, 208, 215, 216, 243, 252
initialize, 17, 25, 70
inner loop, 83

- InputMismatchException, 72
- instance, 161, 172
- instance method, 168, 170, 172
- instance variable, 162, 172
- instantiate, 161, 172
- Integer, 132
- integer division, 20, 21
- interactions, 260
- interface, 252
- interpret, 4, 11
- InterruptedException, 226
- invoke, 45, 56
- IS-A, 215, 216, 223, 240, 250
- iteration, 82, 86, 90
- iterative, 111, 122

J

- JAR, 260, 268
- Java Tutor, 50, 122
- java.awt, 147, 279
- java.io, 29
- java.lang, 31
- java.math, 137
- java.util, 30
- javac, 5
- Javadoc, 153, 259, 271, 275, 278
- JDK, 259, 268
- JFrame, 225, 280
- JVM, 5, 259, 268

K

- keyword, 16, 25, 163

L

- language
 - elements, 31
 - high-level, 4
 - low-level, 4
- leap of faith, 116, 123, 195
- length
 - array, 100
 - string, 86
- library, 29, 40, 152
- Linux, 9
- literal, 33, 40
- local variable, 48, 57
- Logarithm.java, 72
- logic error, 24, 26, 287, 294

- logical operator, 67, 74, 179
- long, 51
- loop, 80, 90
 - for, 81
 - infinite, 80
 - nested, 83, 181
 - search, 183
 - while, 79
- loop body, 80, 90
- loop variable, 83, 90, 98
- low-level language, 4, 11

M

- magic number, 33, 40
- main, 3, 46
- map to, 176
- Math class, 51
- memory, 1, 10
- memory diagram, 18, 26, 87, 99, 130, 147, 163, 169, 177, 189
- mental model, 294
- merge, 194
- merge sort, 192, 200
- method, 3, 10
 - accessor, 166
 - boolean, 70
 - constructor, 162
 - equals, 168, 169
 - getter, 166
 - helper, 191
 - instance, 168, 170
 - modifier, 207
 - mutator, 166
 - parameters, 49
 - setter, 166
 - static, 170
 - toString, 168
 - value, 53
- Mickey Mouse, 282
- MissingFormatArgumentException, 35
- modifier method, 207
- modulo, 37, 41
- modulus, 37, 41
- multidimensional array, 222, 231
- mutable, 150, 155, 157
- mutator, 166

N

- NaN, 74

- NegativeArraySizeException, 96
- neighbor, 220, 227
- nested, 293
 - conditions, 66
 - loops, 83
- nesting, 66, 74, 181
- new, 30, 43, 96, 147, 163
- newline, 6, 11, 112
- NewLine.java, 46
- next
 - Scanner, 72
- nextInt
 - Random, 102
 - Scanner, 33
- nondeterministic, 102, 107
- not operator, 67
- null, 131, 181
- NullPointerException, 131, 135, 292
- NumberFormatException, 133

O

- object, 129, 140
 - array of, 181
 - as parameter, 148
 - displaying, 167
 - mutable, 150
 - type, 161
- Object class, 208
- object code, 5, 11
- object-oriented, 129, 140, 151, 215
- off-by-one, 193, 200
- operand, 19, 26
- operator, 19, 26
 - assignment, 63
 - cast, 36
 - logical, 67, 179
 - new, 30, 43, 96, 147, 163
 - redirection, 263
 - relational, 63
 - remainder, 37
 - string, 22
- or operator, 67
- order of operations, 22, 26, 295
- ordering, 179
- outer loop, 83
- overload, 164, 193
- overloaded, 88
- overloading, 91
- override, 167, 172, 216

P

- package, 29, 40
- paint, 280
- palindrome, 127
- param tag, 274
- parameter, 47, 56, 148
 - multiple, 49
- parameter passing, 48, 56
- parentheses, 22
- parse, 24, 26, 133, 141
- partial ordering, 179
- pi, 51
- Pile, 198
- pixel, 281, 283
- Player, 211
- Point, 147
- polymorphism, 250, 253, 257
- portable, 4, 11
- precedence, 22, 295
- primitive, 129, 140
- print, 6, 167
 - array, 182
 - Card, 177
- print statement, 3, 10, 167, 293, 296
- printDeck, 190
- printf, 34, 89, 167
- println, 3
- PrintTime.java, 49
- PrintTwice.java, 47
- private, 153, 162, 165
- problem solving, 1, 10
- processor, 1, 10
- program, 2, 10
- programming, 3, 10
- prompt, 33, 40
- protected, 239
- pseudocode, 191, 200
- pseudorandom, 102, 108
- public, 45

Q

- quote mark, 6, 84, 288

R

- radians, 51
- RAM, 1
- Random, 102
- randomInt, 191

- rank, 175
- rational number, 174
- Rectangle, 149
- recursion, 114
 - infinite, 163, 291
- recursive, 111, 122
- redirection operator, 263, 268
- reduce, 101, 103, 107
- refactor, 241
- refactoring, 238
- reference, 97, 107, 147, 151, 177, 194
- relational operator, 63, 73
- remainder, 37
- replace, 132
- repository, xii
- return, 53, 73, 150
 - inside loop, 183
- return statement, 295
- return tag, 274
- return type, 53, 57
- return value, 53, 57
- RGB, 281, 283
- rounding error, 21, 26
- row-major order, 223, 231
- rubber duck, 297
- run-time error, 24, 26, 72, 181, 287

S

- scaffolding, 55, 57
- Scanner, 30
- scope, 50, 57, 154
- Scrabble, 93, 173
- search, 101, 107
- selection sort, 192, 200
- semicolon, 3, 23
- sequential search, 183, 185
- setter, 166, 172
- shadowing, 165, 172
- short-circuit evaluation, 68, 74, 136
- shuffle, 190, 191
- signature, 273, 278
- sleep, 226
- sort
 - merge, 192
 - selection, 192
- source code, 5, 11, 152
- specialization, 243, 257
- sprite, 253, 257
- src.zip, 152

- stable configuration, 220
- stack diagram, 50, 57, 113, 115, 120
- stack trace, 35, 41, 292
- StackOverflowError, 113, 163, 290, 292
- state, 17, 26
- statement, 3, 10
 - assignment, 16
 - comment, 4
 - conditional, 64
 - declaration, 15, 147
 - else, 64
 - for, 81
 - if, 64
 - import, 30
 - initialization, 70
 - print, 6, 167, 293, 296
 - return, 53, 73, 150, 183, 295
 - switch, 67
 - while, 79
- Statement
 - catch, 227
 - throw, 248
 - try, 227
- static, 162, 170, 178
- static context, 196, 200
- string, 6, 11
 - array of, 177
 - comparing, 88, 130
 - format, 90, 168
 - length, 86
 - operator, 22
 - reference to, 177
- String class, 147
- StringBuilder, 157, 201
- StringIndexOutOfBoundsException, 86, 136
- stub, 55, 57
- style guide, 7
- subclass, 208, 216
- subdeck, 193
- substring, 87
- suit, 175
- superclass, 208, 216
- Surprise.java, 156
- swapCards, 191
- switch statement, 67
- syntax, 287
- syntax errors, 288
- System.err, 72, 268
- System.in, 30, 129, 263

System.out, 29, 129, 263

T

tag, 274, 278

temporary variable, 53, 57, 295

terminal, 261

testing, 55, 72, 202, 262

text editor, 259, 268

this, 154, 163, 196, 206

Thread.sleep, 226

throw, 248

Time, 161

 addition, 170

toCharArray, 106

token, 32, 40

toLowerCase, 131

top-down design, 191, 200

top-down development, 211, 212

Torvalds, Linus, 9

toString, 149, 168

toUpperCase, 131

tracing, 50, 184, 266, 292

traversal, 100, 107

traverse, 183

try, 227

two-dimensional table, 138

type, 25

 array, 96

 boolean, 63, 69

 char, 15, 84

 double, 20

 int, 15

 long, 22, 51

 object, 161

 String, 6, 15, 147

 void, 45

type cast, 36, 41

U

UML, 153, 157, 196, 215

Unicode, 84, 90

unit test, 267, 269

utility class, 30, 98

V

validate, 71, 74, 135

value, 15, 25

value constructor, 164

value method, 53

variable, 15, 25

 instance, 162

 local, 48

 loop, 83, 98

 private, 153, 162, 165

 static, 178

 temporary, 53, 295

virtual machine, 5, 11, 259

void, 45, 53, 56

W

War (card game), 197

while, 79

wildcard, 264, 269

wrapper class, 132, 141

wrapper method, 201

About the Authors

Allen Downey is a professor of computer science at Olin College of Engineering. He has taught computer science at Wellesley College, Colby College, and the University of California, Berkeley. He has a PhD in computer science from the University of California, Berkeley, and master's and bachelor's degrees from MIT. Allen is the creator of the best-selling Think series for O'Reilly, which includes *Think Python*, *Think Complexity*, *Think DSP*, and *Think Bayes*.

Chris Mayfield is an associate professor of computer science at James Madison University, with a research focus on CS education and professional development. He has a PhD in computer science from Purdue University and bachelor's degrees in CS and German from the University of Utah.

Colophon

The animal on the cover of *Think Java* is a red-tailed black cockatoo (*Calyptorhynchus banksii*), also known as Banks' black cockatoo after an 18th-century English botanist. In the Kunwinjku Aboriginal languages of northern Australia, red-tailed black cockatoos are known as karnamarr. It is a large bird native to Australia, found in many habitats such as forests, open plains, or riverlands, often nesting in eucalyptus trees.

As suggested by their name, these birds have black plumage with a scattering of pale spots, though only males have vivid red panels on their tails (females have orange on their tails). They are typically around 2 feet long and weigh 1–2 pounds. Cockatoos, which branched off about 40 million years ago to form a unique group within the parrot family, are distinguished by their ability to raise a crest of feathers on their heads, as well as the lack of greens and blues in their coloring (colors common in many other parrot species). They also use powder rather than oil glands to keep their feathers clean. But like other parrots, the red-tailed black cockatoo has a large curved beak, and strong feet with two toes facing forward and two facing backward, which allow them to grab and manipulate objects with one foot (essentially using it as humans do hands) while gripping a branch with the other. Some cockatoos, in scientific studies of animal intelligence, have also been shown to use tools. Interestingly, the vast majority of cockatoos are left-footed.

The diet of the red-tailed black cockatoo is primarily made up of eucalyptus seeds, though it will also eat nuts, fruits, insects, and various grains. Like many parrots, they are very noisy and social birds, flocking in large, loud groups of up to 100 birds near plentiful food sources. However, this species is typically very shy around humans.

Red-tailed black cockatoos form strong pair bonds that can last across their long lifetimes (up to 30 years in the wild). They nest in large, old trees, in which they hollow out a cavity where the female lays one egg per year. During incubation and before the

hatchling can fly (usually at three months), both the female and the young bird are fed by the efforts of the male.

Due to its reliance on tall, old trees for nesting, shelter, and food, the red-tailed black cockatoo is sensitive to the same deforestation that threatens other bird and animal populations in southeastern Australia. In addition, while Australia requires a special license to keep and breed these birds, they are still affected by illegal smuggling for the pet trade. Though they have long lifespans in captivity and will bond to a human being, these large, intelligent, far-flying birds are unsuited for a life indoors, and their natural instincts to scream loudly and chew with their large, sharp beaks will render them and their human companions unhappy.

There are more than 20 cockatoo species, all within the larger family of parrots, which live in Australia as well as the island archipelagoes between Southeast Asia and Australia.

Though some of its regional subspecies are considered endangered, the red-tailed black cockatoo is designated as of Least Concern. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from John George Wood's *Illustrated Natural History* (1853). The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning