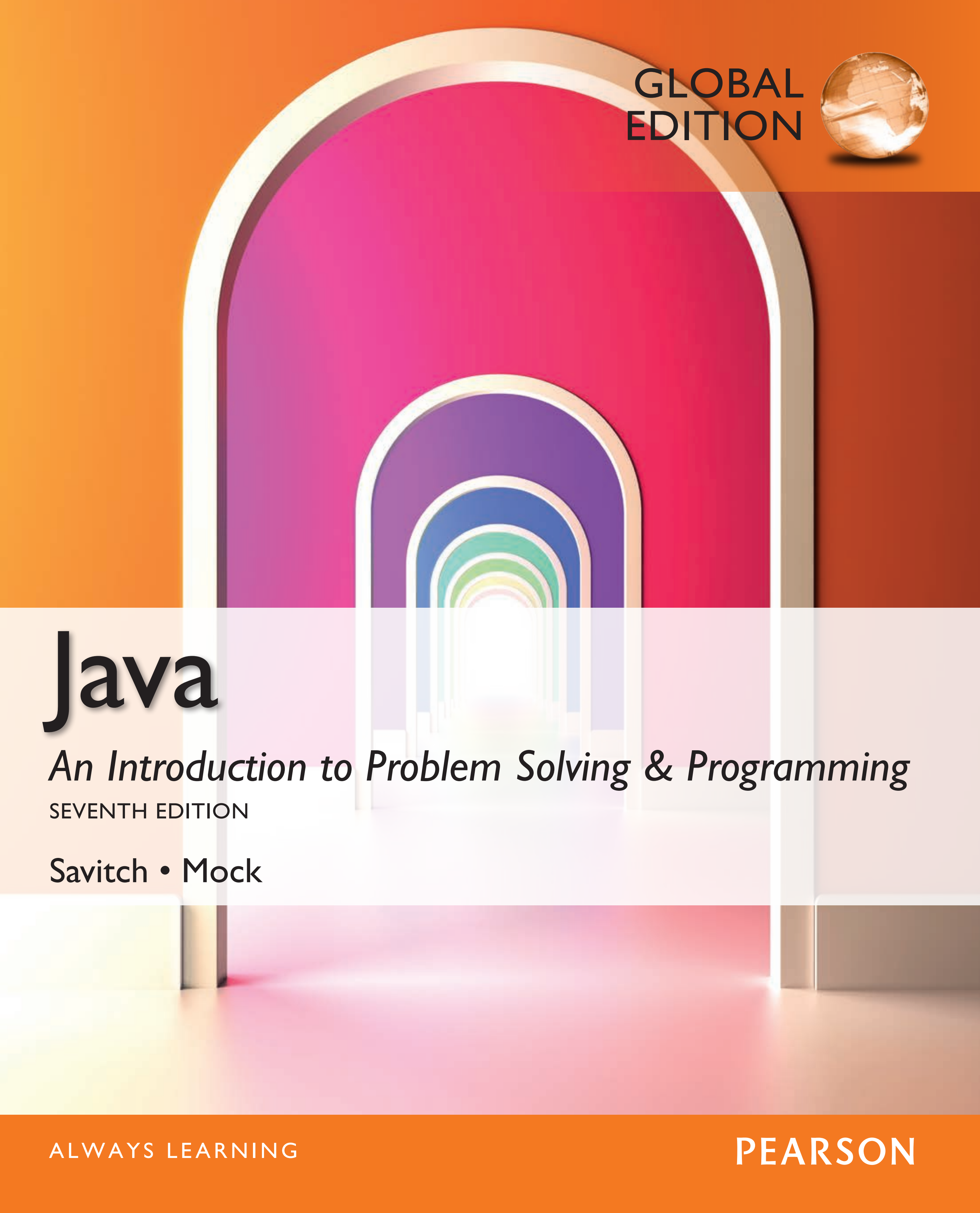


GLOBAL EDITION

Java

An Introduction to Problem Solving & Programming

SEVENTH EDITION

Savitch • Mock

ALWAYS LEARNING

PEARSON

Java™

An Introduction to

Problem Solving & Programming

Java™

7th edition

An Introduction to
Problem Solving & Programming

Global Edition

Walter Savitch
University of California, San Diego

Contributor
Kenrick Mock
University of Alaska Anchorage

Global Edition Contributors
Arup Bhattacharjee
Soumen Mukherjee

PEARSON

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

Editorial Director: *Marcia Horton*
Acquisitions Editor: *Matt Goldstein*
Program Manager: *Kayla Smith-Tarbox*
Editorial Assistant: *Kelsey Loanes*
Marketing Coordinator: *Kathryn Ferranti*
Production Director: *Erin Gregg*
Managing Editor: *Scott Disanno*
Production Project Manager: *Heather McNally*
Head, Learning Asset Acquisition, Global Edition: *Laura Dent*
Acquisitions Editor, Global Edition: *Karthik Subramaniun*
Project Editor, Global Edition: *Anuprova Dey Chowdhuri*
Senior Operations Supervisor: *Vincent Scelta*
Operations Specialist: *Linda Sager*
Art Director: *Kristine Carney*
Cover Designer: Shree Mohanambal Inbakumar
Manager, Rights and Permissions: *Michael Joyce*
Media Director: *Daniel Sandin*
Full-Service Project Management: *Sonam Arora, Cenveo® Publisher Services*
Composition: *Cenveo Publisher Services*
Interior Printer/Bindery: Courier Westford
Cover Printer: Courier Westford

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

ISBN 10: 1292018232
ISBN 13: 978-1-29-201833-1 (Print)
ISBN 13: 978-1-29-206989-0 (PDF)

10 9 8 7 6 5 4 3 2 1
14 13 12 11 10

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

Typeset in 10 GiovanniStd-Book by Cenveo Publisher Services.

Printed and bound by Courier Westford

The publisher's policy is to use paper manufactured from sustainable forests.

Preface for Instructors

Welcome to the seventh edition of *Java: An Introduction to Problem Solving & Programming*. This book is designed for a first course in programming and computer science. It covers programming techniques, as well as the basics of the Java programming language. It is suitable for courses as short as one quarter or as long as a full academic year. No previous programming experience is required, nor is any mathematics, other than a little high school algebra. The book can also be used for a course designed to teach Java to students who have already had another programming course, in which case the first few chapters can be assigned as outside reading.

Changes in This Edition

The following list highlights how this seventh edition differs from the sixth edition:

- End-of-chapter programs are now split into Practice Programs and Programming Projects. Practice Programs require a direct application of concepts presented in the chapter and solutions are usually short. Practice Programs are appropriate for laboratory exercises. Programming Projects require additional problem solving and solutions are generally longer than Practice Programs. Programming Projects are appropriate for homework problems.
- An introduction to functional programming with Java 8's lambda expressions.
- Additional material on secure programming (e.g., overflow, array out of bounds), introduction to Java 2D^{TM} API, networking, and the URL class as further examples of polymorphism in the context of streams.
- Twenty-one new Practice Programs and thirteen new Programming Projects.
- Ten new VideoNotes for a total of seventy two VideoNotes. These VideoNotes walk students through the process of both problem solving and coding to help reinforce key programming concepts. An icon appears in the margin of the book when a VideoNote is available regarding the topic covered in the text.

Latest Java Coverage

All of the code in this book has been tested using a pre-release version of Oracle's Java SE Development Kit (JDK), version 8. Any imported classes are standard and in the Java Class Library that is part of Java. No additional classes or specialized libraries are needed.

Flexibility

If you are an instructor, this book adapts to the way you teach, rather than making you adapt to the book. It does not tightly prescribe the sequence in which your course must cover topics. You can easily change the order in which you teach many chapters and sections. The particulars involved in rearranging material are explained in the dependency chart that follows this preface and in more detail in the "Prerequisites" section at the start of each chapter.

Early Graphics

Graphics supplement sections end each of the first ten chapters. This gives you the option of covering graphics and GUI programming from the start of your course. The graphics supplement sections emphasize applets but also cover GUIs built using the `JFrame` class. Any time after Chapter 8, you can move on to the main chapters on GUI programming (Chapters 13 through 15), which are now on the Web. Alternatively, you can continue through Chapter 10 with a mix of graphics and more traditional programming. Instructors who prefer to postpone the coverage of graphics can postpone or skip the graphics supplement sections.

Coverage of Problem-Solving and Programming Techniques

This book is designed to teach students basic problem-solving and programming techniques and is not simply a book about Java syntax. It contains numerous case studies, programming examples, and programming tips. In addition, many sections explain important problem-solving and programming techniques, such as loop design techniques, debugging techniques, style techniques, abstract data types, and basic object-oriented programming techniques, including UML, event-driven programming, and generic programming using type parameters.

Early Introduction to Classes

Any course that really teaches Java must teach classes early, since everything in Java involves classes. A Java program is a class. The data type for strings of characters is a class. Even the behavior of the equals operator (==) depends on whether it is comparing objects from classes or simpler data items. Classes cannot be avoided, except by means of absurdly long and complicated "magic formulas." This book introduces classes fairly early. Some exposure to using classes is given in Chapters 1 and 2. Chapter 5 covers how to define classes. All of the basic information about classes, including inheritance, is presented by the end of Chapter 8 (even if you omit Chapter 7). However, some topics regarding classes, including inheritance, can be postponed until later in the course.

Although this book introduces classes early, it does not neglect traditional programming techniques, such as top-down design and loop design techniques. These older topics may no longer be glamorous, but they are information that all beginning students need.

Generic Programming

Students are introduced to type parameters when they cover lists in Chapter 12. The class `ArrayList` is presented as an example of how to use a class that has a type parameter. Students are then shown how to define their own classes that include a type parameter.

Language Details and Sample Code

This book teaches programming technique, rather than simply the Java language. However, neither students nor instructors would be satisfied with an introductory programming course that did not also teach the programming language. Until you calm students' fears about language details, it is often impossible to focus their attention on bigger issues. For this reason, the book gives complete explanations of Java language features and lots of sample code. Programs are presented in their entirety, along with sample input and output. In many cases, in addition to the complete examples in the text, extra complete examples are available over the Internet.

Self-Test Questions

Self-test questions are spread throughout each chapter. These questions have a wide range of difficulty levels. Some require only a one-word answer, whereas others require the reader to write an entire, nontrivial program. Complete answers for all the self-test questions, including those requiring full programs, are given at the end of each chapter.

Exercises and Programming Projects

Completely new exercises appear at the end of each chapter. Since only you, and not your students, will have access to their answers, these exercises are suitable for homework. Some could be expanded into programming projects. However, each chapter also contains other programming projects, several of which are new to this edition.

Support Material

The following support materials are available on the Internet at www.pearsonglobaleditions.com/savitch:

For instructors only:

- Solutions to most exercises and programming projects
- PowerPoint slides
- Lab Manual with associated code.

Instructors should click on the registration link and follow instructions to receive a password. If you encounter any problems, please contact your local Pearson Sales Representative. For the name and number of your sales representative, go to www.pearsonglobaleditions.com/savitch.

For students:

- Source code for programs in the book and for extra examples
- Student lab manual
- VideoNotes: video solutions to programming examples and exercises.

Visit www.pearsonglobaleditions.com/savitch to access the student resources.

VideoNote

VideoNotes

VideoNotes are designed for teaching students key programming concepts and techniques. These short step-by-step videos demonstrate how to solve problems from design through coding. VideoNotes allow for self-placed instruction with easy navigation including the ability to select, play, rewind, fast-forward, and stop within each VideoNote exercise.

Margin icons in your textbook let you know when a VideoNote video is available for a particular concept or homework problem.

Integrated Development Environment Resource Kits

Professors who adopt this text can order it for students with a kit containing seven popular Java IDEs (the most recent JDK from Oracle, Eclipse, NetBeans, jGRASP, DrJava, BlueJ, and TextPad). The kit also includes access to a Web site containing written and video tutorials for getting started in each IDE. For ordering information, please contact your campus Pearson Education representative or visit www.pearsonhighered.com.

Contact Us

Your comments, suggestions, questions, and corrections are always welcome. Please e-mail them to savitch.programming.java@gmail.com.

Preface for Students

This book is designed to teach you the Java programming language and, even more importantly, to teach you basic programming techniques. It requires no previous programming experience and no mathematics other than some simple high school algebra. However, to get the full benefit of the book, you should have Java available on your computer, so that you can practice with the examples and techniques given. The latest version of Java is preferable, but a version as early as 5 will do.

If You Have Programmed Before

You need no previous programming experience to use this book. It was designed for beginners. If you happen to have had experience with some other programming language, do not assume that Java is the same as the programming language(s) you are accustomed to using. All languages are different, and the differences, even if small, are large enough to give you problems. Browse the first four chapters, reading at least the Recap portions. By the time you reach Chapter 5, it would be best to read the entire chapter.

If you have programmed before in either C or C++, the transition to Java can be both comfortable and troublesome. At first glance, Java may seem almost the same as C or C++. However, Java is very different from these languages, and you need to be aware of the differences. Appendix 6 compares Java and C++ to help you see what the differences are.

Obtaining a Copy of Java

Appendix 1 provides links to sites for downloading Java compilers and programming environments. For beginners, we recommend Oracle's Java JDK for your Java compiler and related software and TextPad as a simple editor environment for writing Java code. When downloading the Java JDK, be sure to obtain the latest version available.

Support Materials for Students

- Source code for programs in the book and for extra examples
- Student lab manual
- VideoNotes: video solutions to programming examples and exercises.

Visit www.pearsonglobaleditions.com/savitch to access the student resources.

Learning Aids

Each chapter contains several features to help you learn the material:

- The opening overview includes a brief table of contents, chapter objectives and prerequisites, and a paragraph or two about what you will study.
- Recaps concisely summarize major aspects of Java syntax and other important concepts.
- FAQs, or "frequently asked questions," answer questions that other students have asked.
- Remembers highlight important ideas you should keep in mind.
- Programming Tips suggest ways to improve your programming skills.
- Gotchas identify potential mistakes you could make—and should avoid—while programming.
- Asides provide short commentaries on relevant issues.
- Self-Test Questions test your knowledge throughout, with answers given at the end of each chapter. One of the best ways to practice what you are learning is to do the self-test questions *before* you look at the answers.
- A summary of important concepts appears at the end of each chapter.

VideoNote

VideoNotes

These short step-by-step videos demonstrate how to solve problems from design through coding. VideoNotes allow for self-placed instruction with easy navigation including the ability to select, play, rewind, fast-forward, and stop within each VideoNote exercise. Margin icons in your textbook let you know when a VideoNote video is available for a particular concept or homework problem.

This Text Is Also a Reference Book

In addition to using this book as a textbook, you can and should use it as a reference. When you need to check a point that you have forgotten or that you hear mentioned by somebody but have not yet learned yourself, just look in the index. Many index entries give a page number for a "recap." Turn to that page. It will contain a short, highlighted entry giving all the essential points on that topic. You can do this to check details of the Java language as well as details on programming techniques.

Recap sections in every chapter give you a quick summary of the main points in that chapter. Also, a summary of important concepts appears at the end of each chapter. You can use these features to review the chapter or to check details of the Java language.

Acknowledgments

We thank the many people who have made this seventh edition possible, including everyone who has contributed to the first six editions. We begin by recognizing and thanking the people involved in the development of this new edition. The comments and suggestions of the following reviewers were invaluable and are greatly appreciated. In alphabetical order, they are:

Christopher Crick—*Oklahoma State University*
Christopher Plaue—*University of Georgia*
Frank Moore—*University of Alaska Anchorage*
Greg Gagne—*Westminster College*
Helen Hu—*Westminster College*
Paul Bladek—*Edmonds Community College, Washington*
Paul LaFollette—*Temple University*
Pei Wang—*Temple University*
Richard Cassoni—*Palomar College*
Walter Pistone—*Palomar College*

Many other reviewers took the time to read drafts of earlier editions of the book. Their advice continues to benefit this new edition. Thank you once again to:

Adel Elmaghraby—*University of Louisville*
Alan Saleski—*Loyola University Chicago*
Anthony Larrain—*DePaul University*
Arijit Sengupta—*Raj Soin College of Business, Wright State University*
Asa Ben-Hur—*Colorado State University*
Ashraful A. Chowdhury—*Georgia Perimeter College*
Billie Goldstein—*Temple University*
Blayne Mayfield—*Oklahoma State University*
Boyd Trolinger—*Butte College*
Charles Hoot—*Oklahoma City University*
Chris Hoffmann—*University of Massachusetts, Amherst*
Dan Adrian German—*Indiana University*
Dennis Brylow—*Marquette University*
Dolly Samson—*Hawaii Pacific University*
Donald E. Smith—*Rutgers University*
Drew McDermott—*Yale University*
Ed Gellenbeck—*Central Washington University*
Faye Tadayon-Navabi—*Arizona State University*
Gerald Baumgartner—*Louisiana State University*
Gerald H. Meyer—*LaGuardia Community College*
Gobi Gopinath—*Suffolk County Community College*
Gopal Gupta—*University of Texas, Dallas*
H. E. Dunsmore—*Purdue University, Lafayette*
Helen H. Hu—*Westminster College*
Howard Straubing—*Boston College*
James Roberts—*Carnegie Mellon University*
Jim Buffenbarger—*Boise State University*

Joan Boone—*University of North Carolina at Chapel Hill*
John Motil—*California State University, Northridge*
Ken Slonneger—*University of Iowa*
Laird Dornan—*Sun Microsystems, Inc.*
Le Gruenwald—*University of Oklahoma*
Lily Hou—*Carnegie Mellon University*
Liuba Shrira—*Brandeis University*
Martin Chetlen—*Moorpark College*
Mary Elaine Califf—*Illinois State University*
Michele Kleckner—*Elon University*
Michael Clancy—*University of California, Berkeley*
Michael Litman—*Western Illinois University*
Michael Long—*California State University*
Michael Olan—*Richard Stockton College of New Jersey*
Michal Young—*University of Oregon*
Nan C. Schaller—*Rochester Institute of Technology*
Peter Spoerri—*Fairfield University*
Ping-Chu Chu—*Fayetteville State University*
Prasun Dewan—*University of North Carolina, Chapel Hill*
Ricci Heishman—*North Virginia Community College*
Richard Whitehouse—*Arizona State University*
Richard A. Johnson—*Missouri State University*
Richard Ord—*University of California, San Diego*
Robert Herrmann—*Sun Microsystems, Inc., Java Soft*
Robert Holloway—*University of Wisconsin, Madison*
Rob Kelly—*State University of New York at Stony Brook*
Robert P. Burton—*Brigham Young University*
Ryan Shoemaker—*Sun Microsystems, Inc.*
Stan Kwasny—*Washington University*
Stephen F. Weiss—*University of North Carolina, Chapel Hill*
Steven Cater—*Kettering University*
Subramanian Vijayarangam—*University of Massachusetts, Lowell*
Tammy VanDeGrift—*University of Portland*
Thomas Cortina—*Carnegie Mellon University*
Thomas VanDrunen—*Wheaton College*
Y. Annie Liu—*State University of New York at Stony Brook*

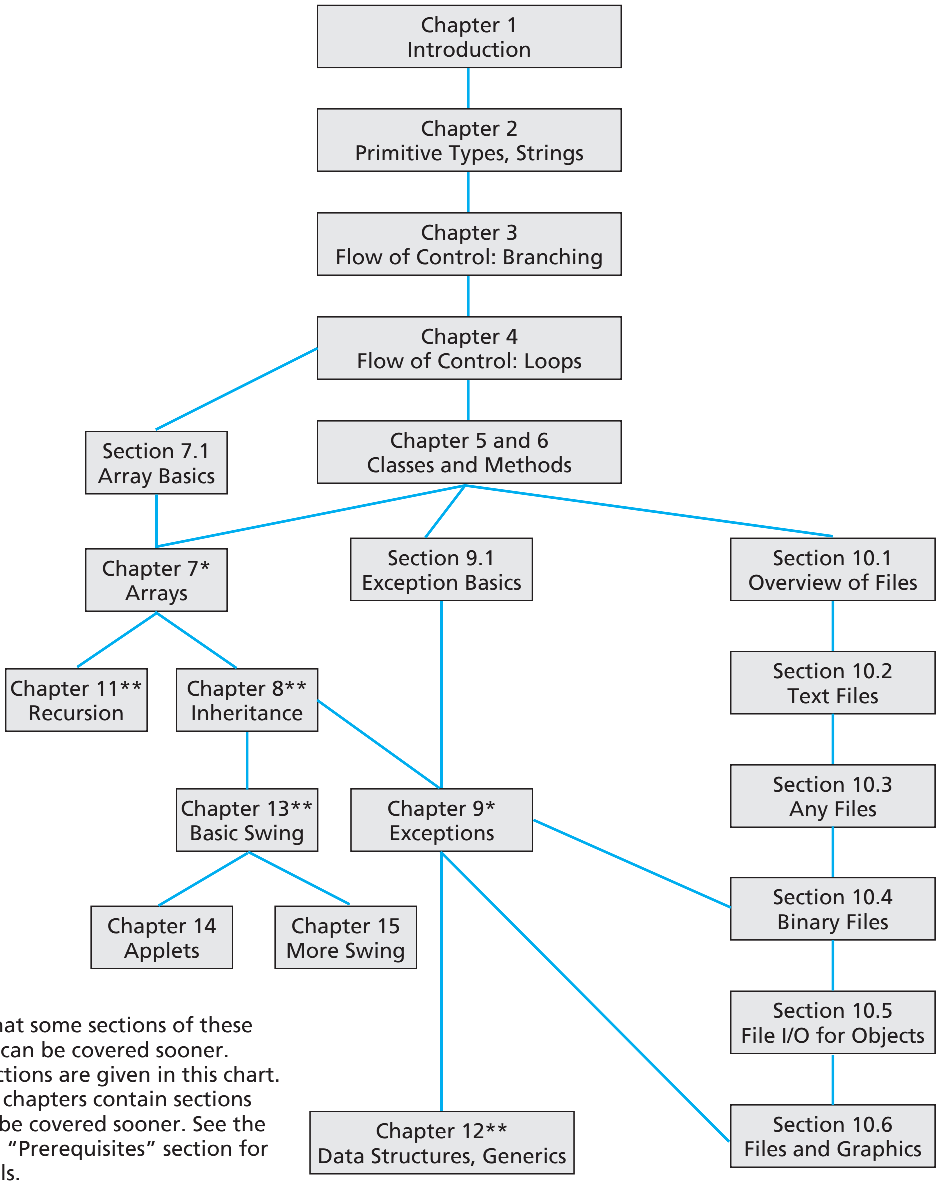
We thank Frank Carrano for his revision of the fifth edition of this textbook. Last but not least, we thank the many students in classes at the University of California, San Diego (UCSD), who were kind enough to help correct preliminary versions of this text, as well as the instructors who class-tested these drafts. In particular, we extend a special thanks to Carole McNamee of California State University, Sacramento, and to Paul Kube of UCSD. These student comments and the detailed feedback and class testing of earlier editions of the book were a tremendous help in shaping the final book.

W. S.
K. M.

Pearson Education wishes to thank B.R. Chandravarkar of NITK, Arup Bhattacharjee, Soumen Mukherjee, and Raghavan, for reviewing the Global Edition.

Dependency Chart

This chart shows the prerequisites for the chapters in the book. If there is a line between two boxes, the material in the higher box should be covered before the material in the lower box. Minor variations to this chart are discussed in the "Prerequisites" section at the start of each chapter. These variations usually provide more, rather than less, flexibility than what is shown on the chart.



* Note that some sections of these chapters can be covered sooner. Those sections are given in this chart.
** These chapters contain sections that can be covered sooner. See the chapter's "Prerequisites" section for full details.

Features of This Text

Recaps

Summarize Java syntax and other important concepts.

RECAP Bytes and Memory Locations

A computer's main memory is divided into numbered units called bytes. The number of a byte is called its address. Each byte can hold eight binary digits, or bits
piece of data that is too l
uses several adjacent byte
single, larger memory loc
of the adjacent bytes.

Remembers

Highlight important ideas that students should keep in mind.

REMEMBER Syntactic Variables

When you see something in this book like *Type*, *Variable_1*, or *Variable_2* used to describe Java syntax, these words do not literally appear in your Java code. They are **syntactic variables,** which are a kind of blank that you fill in with something from the category that they describe. For example, *Type* can be replaced by `int`, `double`, `char`, or any other type name. *Variable_1* and *Variable_2* can each be replaced by any variable name.

Programming Tips

Give students helpful advice about programming in Java.

PROGRAMMING TIP Initialize Variables

A variable that has been declared, but that has not yet been given a value by an assignment statement (or in some other way), is said to be **uninitialized.** If the variable is a variable of a class type, it literally has no value. If the variable has a primitive type, it likely has some default value. However, your program will be clearer if you explicitly give the variable a value, even if you are simply reassigning the default value. (The exact details on default values have been known to change and should not be counted on.)

One easy way to ensure that you do not have an uninitialized variable is to initialize it within the declaration. Simply combine the declaration and an assignment statement, as in the following examples:

```
int count = 0;
double taxRate = 0.075;
char grade = 'A';
int balance = 1000, newBalance;
```

Note that you can initialize some variables and not initialize others in a declaration.

Sometimes the compiler may complain that you have failed to initialize a variable. In most cases, that will indeed be true. Occasionally, though, the compiler is mistaken in giving this advice. However, the compiler will not compile your program until you convince it that the variable in question is initialized. To make the compiler happy, initialize the variable when you declare it, even if the variable will be given another value before it is used for anything. In such cases, you cannot argue with the compiler. ■

Gotchas

Identify potential mistakes in programming that students might make and should avoid.

GOTCHA Hidden Errors

Just because your program compiles and runs without any errors and even produces reasonable-looking output does not mean that your program is correct. You should always run your program with some test data that gives predictable output. To do this, choose some data for which you can compute the correct results, either by using pencil and paper, by looking up the answer, or by some other means. Even this testing does not guarantee that your program is correct, but the more testing you do, the more confidence you can have in your program. ■

FAQs

Provide students answers to frequently asked questions within the context of the chapter.

FAQ[11] FAQ stands for "frequently asked question." **Why just 0s and 1s?**

Computers use 0s and 1s because it is easy to make an electrical device that has only two stable states. However, when you are programming, you normally need not be concerned about the encoding of data as 0s and 1s. You can program as if the computer directly stored numbers, letters, or strings of characters in memory.

There is nothing special about calling the states *zero* and *one*. We could just as well use any two names, such as *A* and *B* or *true* and *false*. The important thing is that the underlying physical device has two stable states, such as on and off or high voltage and low voltage. Calling these two states *zero* and *one* is simply a convention, but it's one that is almost universally followed.

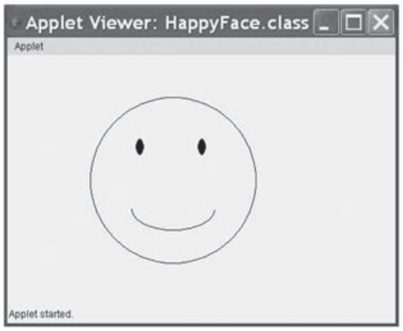
Listings

Show students complete programs with sample output.

LISTING 1.2 Drawing a Happy Face

```
import javax.swing.JApplet;
import java.awt.Graphics;
public class HappyFace extends JApplet
{
   public void paint(Graphics canvas)
   {
      canvas.drawOval(100, 50, 200, 200);
      canvas.fillOval(155, 100, 10, 20);
      canvas.fillOval(230, 100, 10, 20);
      canvas.drawArc(150, 160, 100, 50, 180, 180);
   }
}
```

Applet Output



Case Studies

Take students from problem statement to algorithm development to Java code.

CASE STUDY Unit Testing

So far we've tested our programs by running them, typing in some input, and visually checking the results to see if the output is what we expected. This is fine for small programs but is generally insufficient for large programs. In a large program there are usually so many combinations of interacting inputs that it would take too much time to manually verify the correct result for all inputs. Additionally, it is possible that code changes result in unintended side effects. For example, a fix for one error might introduce a different error. One way to attack this problem is to write **unit tests.** Unit testing is a methodology in which the programmer tests the correctness of individual units of code. A unit is often a method but it could be a class or other group of code.

The collection of unit tests becomes the **test suite.** Each test is generally automated so that human input is not required. Automation is important because it is desirable to have tests that run often and quickly. This makes it possible to run the tests repeatedly, perhaps once a day or every time code is changed, to make sure that everything is still working. The process of running tests repeatedly is called **regression testing.**

Let's start with a simple test case for the `Species` class in Listing 5.19. Our first test might be to verify that the name, initial population, and growth rate is correctly set in the `setSpecies` method. We can accomplish this by creating

VideoNotes

Step-by-step video solutions to programming examples and homework exercises.

VideoNote
Writing arithmetic expressions and statements

Programming Examples

Provide more examples of Java programs that solve specific problems.

PROGRAMMING EXAMPLE Nested Loops

The body of a loop can contain any sort of statements. In particular, you can have a loop statement within the body of a larger loop statement. For example, the program in Listing 4.4 uses a `while` loop to compute the average of a list of nonnegative scores. The program asks the user to enter all the scores followed by a negative sentinel value to mark the end of the data. This `while` loop is placed inside a `do-while` loop so that the user can repeat the entire process for another exam, and another, until the user wishes to end the program.

Self-Test Questions

Provide students with the opportunity to practice skills learned in the chapter. Answers at the end of each chapter give immediate feedback.

SELF-TEST QUESTIONS

28. Given the class `Species` as defined in Listing 5.19, why does the following program cause an error message?

```
    public class SpeciesEqualsDemo
    {
    public static void main(String[] args)
    {
        Species s1, s2; s1.
        setSpecies("Klingon ox", 10, 15);
        s2.setSpecies("Klingon ox", 10, 15);
        if (s1 == s2)
            System.out.println("Match with ==.");
        else
            System.out.println("Do Notmatchwith ==.")
    }
}
```

29. After correcting the program in the previous question, what output does the program produce?

30. What is the biggest difference between a parameter of a primitive type and a parameter of a class type?

31. Given the class `Species`, as defined in Listing 5.19, and the class

Asides

Give short commentary on relevant topics.

ASIDE Use of the Terms *Parameter* and *Argument*

Our use of the terms *parameter* and *argument* is consistent with common usage. We use *parameter* to describe the definition of the data type and variable inside the header of a method and *argument* to describe items passed into a method when it is invoked. However, people often use these terms interchangeably. Some people use the term *parameter* both for what we call a *formal parameter* and for what we call an *argument*. Other people use the term *argument* both for what we call a *formal parameter* and for what we call an *argument*. When you see the term *parameter* or *argument* in other books, you must figure out its exact meaning from the context.

Brief Contents

The following chapters and appendices, along with an index to their contents, are on the book's Web site:

Contents

This page is intentionally left blank.

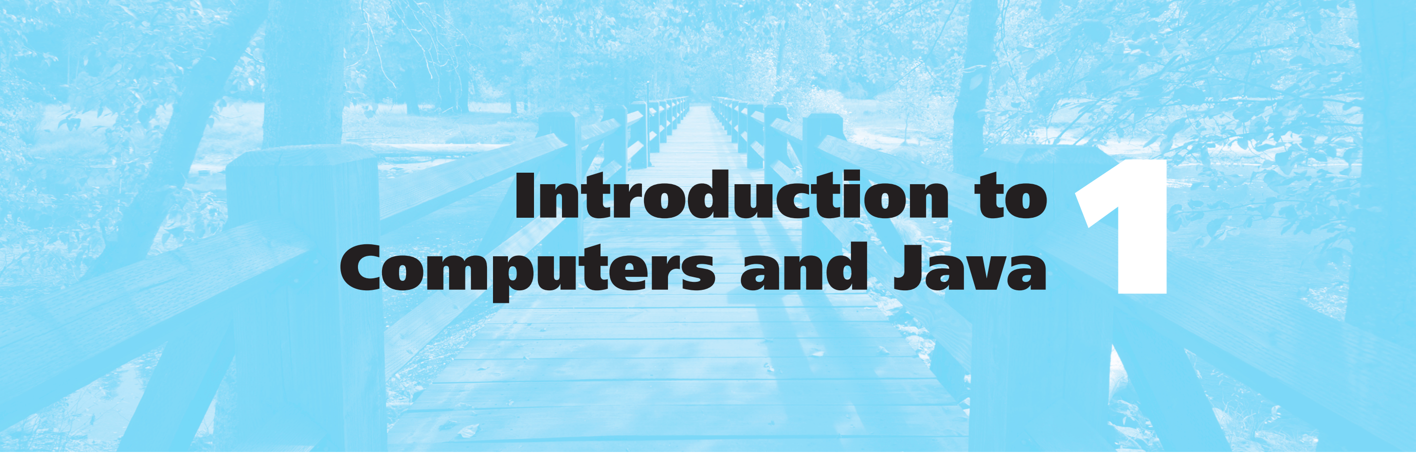
LOCATION OF VIDEONOTES IN THE TEXT

This page is intentionally left blank.

LOCATION OF VIDEONOTES IN THE TEXT

Chapter 7	Using arrays within a class, p. 549 Using arrays within methods, p. 560 The `java.util.Arrays` class, p. 577 Coding with two-dimensional arrays, p. 587 Inserting into a sorted array, p. 612 Solution to Project 7, p. 613
Chapter 8	Protected instance variables and methods, p. 632 Defining classes using inheritance, p. 644 Exploring polymorphism, p. 654 Exploring interfaces, p. 665 Defining an abstract class, p. 686 Deriving a class from `Vehicle`, p. 696
Chapter 9	Using predefined exception classes, p. 718 Using your exception classes, p. 726 Using a `throws` clause, p. 741 Solution to Practice Program 2, p. 763 Defining an exception class, p. 766
Chapter 10	Writing and reading a text file, p. 788 Writing and reading a binary file, p. 817 Using a binary file of objects and arrays, p. 827 Networking with Streams, p. 827 Processing a file, p. 843 Solution to Practice Program 3, p. 844
Chapter 11	Writing `void` methods that are recursive, p. 867 Recursion and stack overflow, p. 869 Writing recursive methods that return a value, p. 875 Using recursion to search and sort an array, p. 878 Solution to Project 1, p. 894 Drawing recursively, p. 899
Chapter 12	Walkthrough of the `HashMap` demonstration, p. 918 Using `HashMap` with a custom class, p. 920 Using `List`, `ArrayList`, and `LinkedList`, p. 922 Adding a node anywhere in a linked list, p. 934 Creating classes that use generics, p. 957 Defining a circular linked list, p. 966 Solution to Project 14, p. 969

This page is intentionally left blank.



Introduction to Computers and Java 1

It is by no means hopeless to expect to make a machine for really very difficult mathematical problems. But you would have to proceed step-by-step. I think electricity would be the best thing to rely on.

—CHARLES SANDERS PEIRCE (1839–1914)

INTRODUCTION

This chapter gives you a brief overview of computer hardware and software. Much of this introductory material applies to programming in any language, not just to programming in Java. Our discussion of software will include a description of a methodology for designing programs known as object-oriented programming. Section 1.2 introduces the Java language and explains a sample Java program.

Section 1.4 is the first of a number of graphics supplements that end each of the first ten chapters and provide an introduction to the graphics capabilities of the Java language. These graphics supplements are interdependent, and each one uses the Java topics presented in its chapter.

OBJECTIVES

After studying this chapter, you should be able to

- Give a brief overview of computer hardware and software
- Give an overview of the Java programming language
- Describe the basic techniques of program design in general and object-oriented programming in particular
- Describe applets and some graphics basics

PREREQUISITES

This first chapter does *not* assume that you have had any previous programming experience, but it does assume that you have access to a computer. To get the full value from the chapter, and from the rest of this book, you should have a computer that has the Java language installed, so that you can try out what you are learning. Appendix 1 describes how to obtain and install a free copy of the Java language for your computer.

1.1 COMPUTER BASICS

The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. Its province is to assist us in making available what we are already acquainted with.

—ADA AUGUSTA, *Countess of Lovelace* (1815–1852)

Computer systems consist of hardware and software. The **hardware** is the physical machine. A set of instructions for the computer to carry out is called a **program.** All the different kinds of programs used to give instructions to the computer are collectively referred to as **software.** In this book, we will discuss software, but to understand software, it helps to know a few basic things about computer hardware.

Hardware and software make up a computer system

Hardware and Memory

Most computers available today have the same basic components, configured in basically the same way. They all have input devices, such as a keyboard and a mouse. They all have output devices, such as a display screen and a printer. They also have several other basic components, usually housed in some sort of cabinet, where they are not so obvious. These other components store data and perform the actual computing.

The **CPU,** or **central processing unit,** or simply the **processor,** is the device inside your computer that follows a program's instructions. Currently, one of the better-known processors is the Intel®Core™i7 processor. The processor can carry out only very simple instructions, such as moving numbers or other data from one place in memory to another and performing some basic arithmetic operations like addition and subtraction. The power of a computer comes from its speed and the intricacies of its programs. The basic design of the hardware is conceptually simple.

The CPU, or central processing unit, or processor, performs the instructions in a program

A computer's **memory** holds **data** for the computer to process, and it holds the result of the computer's intermediate calculations. Memory exists in two basic forms, known as main memory and auxiliary memory. **Main memory** holds the current program and much of the data that the program is manipulating. You most need to be aware of the nature of the main memory when you are writing programs. The information stored in main memory typically is **volatile,** that is, it disappears when you shut down your computer. In contrast, the data in **auxiliary memory,** or **secondary memory,** exists even when the computer's power is off. All of the various kinds of disks—including hard disk drives, flash drives, compact discs (CDs), and digital video discs (DVDs) are auxiliary memory.

Main memory is volatile; auxiliary memory is not

To make this more concrete, let's look at an example. You might have heard a description of a personal computer (PC) as having, say, 1 gigabyte of RAM and a 200-gigabyte hard drive. **RAM**—short for **random access memory**—is the main memory, and the hard drive is the principal—but not the only—form of auxiliary memory. A byte is a quantity of memory. So 1 gigabyte of RAM is approximately 1 billion bytes of memory, and a 200-gigabyte hard drive has approximately 200 billion bytes of memory. What exactly is a byte? Read on.

The computer's main memory consists of a long list of numbered bytes. The number of a byte is called its **address.** A **byte** is the smallest addressable unit of memory. A piece of data, such as a number or a keyboard character, can be stored in one of these bytes. When the computer needs to recover the data later, it uses the address of the byte to find the data item.

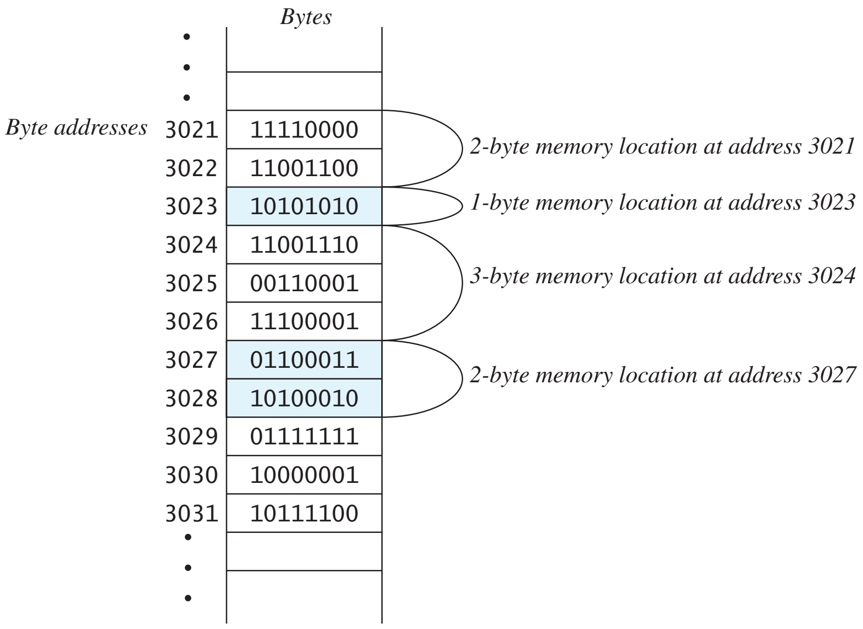
Main memory consists of addressable eight-bit bytes

A byte, by convention, contains eight digits, each of which is either 0 or 1. Actually, any two values will do, but the two values are typically written as 0 and 1. Each of these digits is called a **binary digit** or, more typically, a **bit.** A byte, then, contains eight bits of memory. Both main memory and auxiliary memory are measured in bytes.

Data of various kinds, such as numbers, letters, and strings of characters, is encoded as a series of 0s and 1s and placed in the computer's memory. As it turns out, one byte is just large enough to store a single keyboard character. This is one of the reasons that a computer's memory is divided into these eight-bit bytes instead of into pieces of some other size. However, storing either a string of characters or a large number requires more than a single byte. When the computer needs to store a piece of data that cannot fit into a single byte, it uses several adjacent bytes. These adjacent bytes are then considered to be a single, larger **memory location,** and the address of the first byte is used as the address of the entire memory location. Figure 1.1 shows how a typical computer's main memory might be divided into memory locations. The addresses of these larger locations are not fixed by the hardware but depend on the program using the memory.

Groups of adjacent bytes can serve as a single memory location

FIGURE 1.1 Main Memory



Recall that main memory holds the current program and much of its data. Auxiliary memory is used to hold data in a more or less permanent form. Auxiliary memory is also divided into bytes, but these bytes are grouped into much larger units known as **files.** A file can contain almost any sort of data, such as a program, an essay, a list of numbers, or a picture, each in an encoded form. For example, when you write a Java program, you will store the program in a file that will typically reside in some kind of disk storage. When you use the program, the contents of the program file are copied from auxiliary memory to main memory.

A file is a group of bytes stored in auxiliary memory

You name each file and can organize groups of files into **directories,** or **folders.** *Folder* and *directory* are two names for the same thing. Some computer systems use one name, and some use the other.

A directory, or folder, contains groups of files

FAQ[1] Why just 0s and 1s?

Computers use 0s and 1s because it is easy to make an electrical device that has only two stable states. However, when you are programming, you normally need not be concerned about the encoding of data as 0s and 1s. You can program as if the computer directly stored numbers, letters, or strings of characters in memory.

There is nothing special about calling the states *zero* and *one*. We could just as well use any two names, such as *A* and *B* or *true* and *false*. The important thing is that the underlying physical device has two stable states, such as on and off or high voltage and low voltage. Calling these two states *zero* and *one* is simply a convention, but it's one that is almost universally followed.

RECAP Bytes and Memory Locations

A computer's main memory is divided into numbered units called bytes. The number of a byte is called its address. Each byte can hold eight binary digits, or bits, each of which is either 0 or 1. To store a piece of data that is too large to fit into a single byte, the computer uses several adjacent bytes. These adjacent bytes are thought of as a single, larger memory location whose address is the address of the first of the adjacent bytes.

[1] FAQ stands for "frequently asked question."

Programs

A program is a set of computer instructions

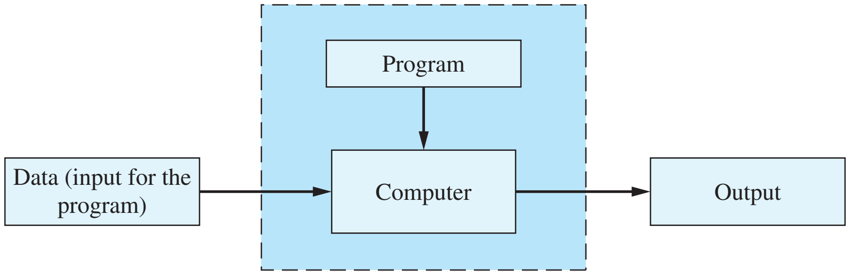
You probably have some idea of what a program is. You use programs all the time. For example, text editors and word processors are programs. As we mentioned earlier, a program is simply a set of instructions for a computer to follow. When you give the computer a program and some data and tell the computer to follow the instructions in the program, you are **running,** or **executing,** the program on the data.

Figure 1.2 shows two ways to view the running of a program. To see the first way, ignore the dashed lines and blue shading that form a box. What's left is what really happens when you run a program. In this view, the computer has two kinds of input. The program is one kind of input; it contains the instructions that the computer will follow. The other kind of input is the data for the program. It is the information that the computer program will process. For example, if the program is a spelling-check program, the data would be the text that needs to be checked. As far as the computer is concerned, both the data and the program itself are input. The output is the result—or results—produced when the computer follows the program's instructions. If the program checks the spelling of some text, the output might be a list of words that are misspelled.

This first view of running a program is what really happens, but it is not always the way we think about running a program. Another way is to think of the data as the input to the program. In this second view, the computer and the program are considered to be one unit. Figure 1.2 illustrates this view by surrounding the combined program–computer unit with a dashed box and blue shading. When we take this view, we think of the data as input to the program and the results as output from the program. Although the computer is understood to be there, it is presumed just to be something that assists the program. People who write programs—that is, **programmers**—find this second view to be more useful when they design a program.

Your computer has more programs than you might think. Much of what you consider to be "the computer" is actually a program—that is, software—rather than hardware. When you first turn on a computer, you are already

FIGURE 1.2 Running a Program



running and interacting with a program. That program is called the **operating system.** The operating system is a kind of supervisory program that oversees the entire operation of the computer. If you want to run a program, you tell the operating system what you want to do. The operating system then retrieves and starts the program. The program you run might be a text editor, a browser to surf the World Wide Web, or some program that you wrote using the Java language. You might tell the operating system to run the program by using a mouse to click an icon, by choosing a menu item, or by typing in a simple command. Thus, what you probably think of as "the computer" is really the operating system. Some common operating systems are Microsoft Windows, Apple's (Macintosh) Mac OS, Linux, and UNIX.

An operating system is a program that supervises a computer's operation

FAQ What exactly is software?

The word *software* simply means programs. Thus, a software company is a company that produces programs. The software on your computer is just the collection of programs on your computer.

Programming Languages, Compilers, and Interpreters

Most modern programming languages are designed to be relatively easy for people to understand and use. Such languages are called **high-level languages.** Java is a high-level language. Most other familiar programming languages, such as Visual Basic, C++, C#, COBOL, Python, and Ruby, are also high-level languages. Unfortunately, computer hardware does not understand high-level languages. Before a program written in a high-level language can be run, it must be translated into a language that the computer can understand.

Java is a high-level language

The language that the computer can directly understand is called **machine language. Assembly language** is a symbolic form of machine language that is easier for people to read. So assembly language is almost the same thing as machine language, but it needs some minor additional translation before it can run on the computer. Such languages are called **low-level languages.**

Computers execute a low-level language called machine language

The translation of a program from a high-level language, like Java, to a low-level language is performed entirely or in part by another program. For some high-level languages, this translation is done as a separate step by a program known as a **compiler.** So before you run a program written in a high-level language, you must first run the compiler on the program. When you do this, you are said to **compile** the program. After this step, you can run the resulting machine-language program as often as you like without compiling it again.

Compile once, execute often

The terminology here can get a bit confusing, because both the input to the compiler program and the output from the compiler program are programs. Everything in sight is a program of some kind or other. To help

Compilers translate source code into object code

avoid confusion, we call the input program, which in our case will be a Java program, the **source program,** or **source code.** The machine-language program that the compiler produces is often called the **object program,** or **object code.** The word **code** here just means a program or a part of a program.

RECAP Compiler

A compiler is a program that translates a program written in a high-level language, such as Java, into a program in a simpler language that the computer can more or less directly understand.

Interpreters translate and execute portions of code at a time

Some high-level languages are translated not by compilers but rather by another kind of program called an **interpreter.** Like a compiler, an interpreter translates program statements from a high-level language to a low-level language. But unlike a compiler, an interpreter executes a portion of code right after translating it, rather than translating the entire program at once. Using an interpreter means that when you run a program, translation alternates with execution. Moreover, translation is done each time you run the program. Recall that compilation is done once, and the resulting object program can be run over and over again without engaging the compiler again. This implies that a compiled program generally runs faster than an interpreted one.

RECAP Interpreter

An interpreter is a program that alternates the translation and execution of statements in a program written in a high-level language.

One disadvantage of the processes we just described for translating programs written in most high-level programming languages is that you need a different compiler or interpreter for each type of language or computer system. If you want to run your source program on three different types of computer systems, you need to use three different compilers or interpreters. Moreover, if a manufacturer produces an entirely new type of computer system, a team of programmers must write a new compiler or interpreter for that computer system. This is a problem, because these compilers and interpreters are large programs that are expensive and time-consuming to write. Despite this cost, many high-level-language compilers and interpreters work this way. Java, however, uses a slightly different and much more versatile approach that combines a compiler and an interpreter. We describe Java's approach next.

Java Bytecode

The Java compiler does not translate your program into the machine language for your particular computer. Instead, it translates your Java program into a language called **bytecode.** Bytecode is not the machine language for any particular computer. Instead, bytecode is a machine language for a hypothetical computer known as a **virtual machine.** A virtual machine is not exactly like any particular computer, but is similar to all typical computers. Translating a program written in bytecode into a machine-language program for an actual computer is quite easy. The program that does this translation is a kind of interpreter called the **Java Virtual Machine,** or **JVM.** The JVM translates and runs the Java bytecode.

A compiler translates Java code into bytecode

To run your Java program on your computer, you proceed as follows: First, you use the compiler to translate your Java program into bytecode. Then you use the particular JVM for your computer system to translate each bytecode instruction into machine language and to run the machine-language instructions. The whole process is shown in Figure 1.3.

The JVM is an interpreter that translates and executes bytecode

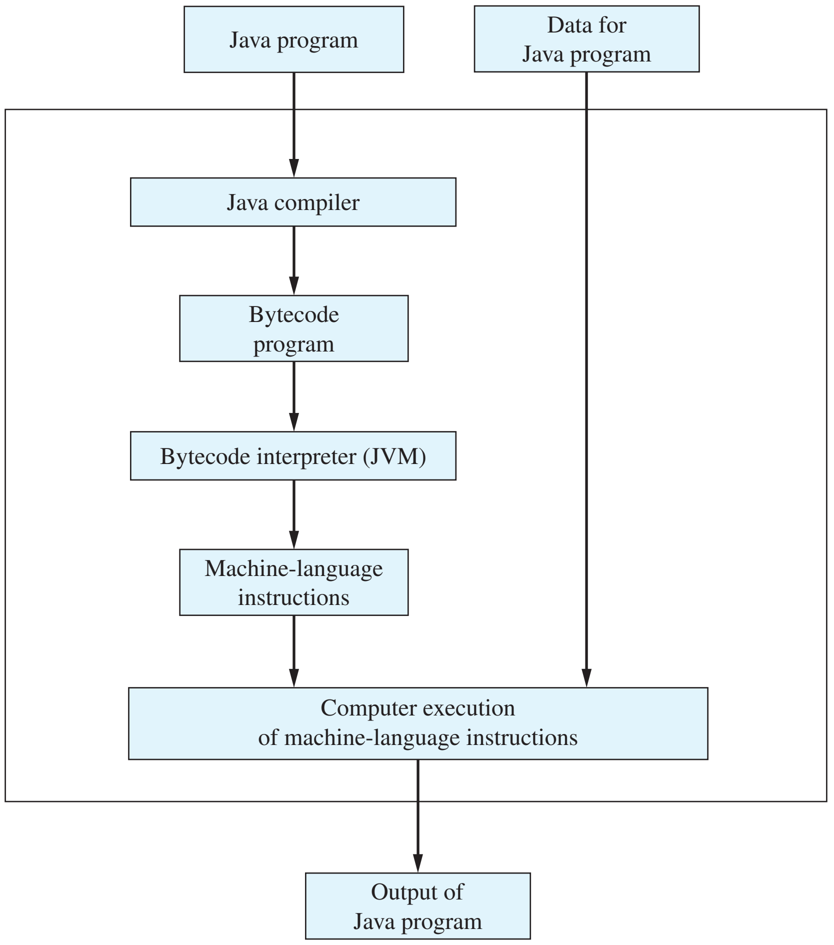
Modern implementations of the JVM use a Just-in-Time (JIT), compiler. The JIT compiler reads the bytecode in chunks and compiles entire chunks to native machine language instructions as needed. The compiled machine language instructions are remembered for future use so a chunk needs to be compiled only once. This model generally runs programs faster than the interpreter model, which repeatedly translates the next bytecode instruction to machine code.

It sounds as though Java bytecode just adds an extra step to the process. Why not write compilers that translate directly from Java to the machine language for your particular computer system? That could be done, and it is what is done for many other programming languages. Moreover, that technique would produce machine-language programs that typically run faster. However, Java bytecode gives Java one important advantage, namely, portability. After you compile your Java program into bytecode, you can run that bytecode on any computer. When you run your program on another computer, you do not need to recompile it. This means that you can send your bytecode over the Internet to another computer and have it run easily on that computer regardless of the computer's operating system. That is one of the reasons Java is good for Internet applications.

Java bytecode runs on any computer that has a JVM

Portability has other advantages as well. When a manufacturer produces a new type of computer system, the creators of Java do not have to design a new Java compiler. One Java compiler works on every computer. Of course, every type of computer must have its own bytecode interpreter—the JVM—that translates bytecode instructions into machine-language instructions for that particular computer, but these interpreters are simple programs compared to a compiler. Thus, Java can be added to a new computer system very quickly and very economically.

FIGURE 1.3 Compiling and Running a Java Program


Java program
Data for Java program
Java compiler
Bytecode program
Bytecode interpreter (JVM)
Machine-language instructions
Computer execution of machine-language instructions
Output of Java program

RECAP Bytecode

The Java compiler translates your Java program into a language called bytecode. This bytecode is not the machine language for any particular computer, but it is similar to the machine language of most common computers. Bytecode is easily translated into the machine language of a given computer. Each type of computer will have its own translator—called an interpreter—that translates from bytecode instructions to machine-language instructions for that computer.

Knowing about Java bytecode is important, but in the day-to-day business of programming, you will not even be aware that it exists. You normally will give two commands, one to compile your Java program into bytecode and one to run your program. The **run command** tells the bytecode interpreter to execute the bytecode. This run command might be called "run" or something else, but it is unlikely to be called "interpret." You will come to think of the run command as running whatever the compiler produces, and you will not even think about the translation of bytecode to machine language.

FAQ Why is it called bytecode?

Programs in low-level languages, such as bytecode and machine-language code, consist of instructions, each of which can be stored in a few bytes of memory. Typically, one byte of each instruction contains the operation code, or opcode, which specifies the operation to be performed. The notion of a one-byte opcode gave rise to the term *bytecode*.

Class Loader

A Java program is seldom written as one piece of code all in one file. Instead, it typically consists of different pieces, known as **classes.** We will talk about classes in detail later, but thinking of them as pieces of code is sufficient for now. These classes are often written by different people, and each class is compiled separately. Thus, each class is translated into a different piece of bytecode. To run your program, the bytecode for these various classes must be connected together. The connecting is done by a program known as the **class loader.** This connecting is typically done automatically, so you normally need not be concerned with it. In other programming languages, the program corresponding to the Java class loader is called a *linker*.

For now, think of a class as a piece of code

SELF-TEST QUESTIONS

Answers to the self-test questions appear at the end of each chapter.

1. What are the two kinds of memory in a computer?

2. What is software?

3. What data would you give to a program that computes the sum of two numbers?

4. What data would you give to a program that computes the average of all the quizzes you have taken in a course?

5. What is the difference between a program written in a high-level language, a program in machine language, and a program expressed in Java bytecode?

6. Is Java a high-level language or a low-level language?

7. Is Java bytecode a high-level language or a low-level language?

8. What is a compiler?

9. What is a source program?

10. What do you call a program that translates Java bytecode into machine-language instructions?

1.2 A SIP OF JAVA

"New Amsterdam, madame," replied the Prince, "and after that the Sunda Islands and beautiful Java with its sun and palm trees."

—HENRY DE VERE STACPOOLE, *The Beach of Dreams*

In this section, we describe some of the characteristics of the Java language and examine a simple Java program. This introduction is simply an overview and a presentation of some terminology. We will begin to explore the details of Java in the next chapter.

History of the Java Language

Java is widely viewed as a programming language for Internet applications. However, this book, and many other people, views Java as a general-purpose programming language that can be used without any reference to the Internet. At its birth, Java was neither of these things, but it eventually evolved into both.

The history of Java goes back to 1991, when James Gosling and his team at Sun Microsystems began designing the first version of a new programming language that would become Java—though it was not yet called that. This new language was intended for programming home appliances, like toasters and TVs. That sounds like a humble engineering task, but in fact it's a very challenging one. Home appliances are controlled by a wide variety of computer processors (chips). The language that Gosling and his team were designing had to work on all of these different processors. Moreover, a home appliance is typically an inexpensive item, so no manufacturer would be willing to invest large amounts of time and money into developing complicated compilers to translate the appliance-language programs into a language the processor could understand. To solve these challenges, the designers wrote one piece of software that would translate an appliance-language program into a program in an intermediate language that would be the same for all appliances and their processors. Then a small, easy-to-write and hence inexpensive program would translate the intermediate language into the machine language for a particular appliance or computer. The intermediate language was called bytecode. The plan for programming appliances using this first version of Java never caught on with appliance manufacturers, but that was not the end of the story.

In 1994, Gosling realized that his language—now called Java—would be ideal for developing a Web browser that could run programs over the Internet. The Web browser was produced by Patrick Naughton and Jonathan Payne at Sun Microsystems. Originally called WebRunner and then HotJava, this browser is no longer supported. But that was the start of Java's connection to the Internet. In the fall of 1995, Netscape Communications Corporation decided to make the next release of its Web browser capable of running Java programs. Other companies associated with the Internet followed suit and have developed software that accommodates Java programs.

FAQ Why is the language named Java?

The question of how Java got its name does not have a very interesting answer. The current custom is to name programming languages in pretty much the same way that parents name their children. The creator of the programming language simply chooses any name that sounds good to her or him. The original name of the Java language was Oak. Later the creators realized that there already was a computer language named Oak, so they needed another name, and Java was chosen. One hears conflicting explanations of the origin of the name Java. One traditional, and perhaps believable, story is that the name was thought of during a long and tedious meeting while the participants drank coffee, and the rest, as they say, is history.

Applications and Applets

This book focuses on two kinds of Java programs: applications and applets. An **application** is just a regular program. An **applet** sounds as though it would be a little apple, but the name is meant to convey the idea of a little application. Applets and applications are almost identical. The difference is that an application is meant to be run on your computer, like any other program, whereas an applet is meant to be sent to another location on the Internet and run there.

Applications are regular programs

Applets run within a Web browser

Once you know how to design and write one of these two kinds of programs, either applets or applications, it is easy to learn to write the other kind. This book is organized to allow you to place as much or as little emphasis on applets as you wish. In most chapters the emphasis is on applications, but the graphics supplements at the ends of the first ten chapters give material on applets. Applets are also covered in detail in Chapter 14, which is on the book's Web site. You may choose to learn about applets along the way, by doing the graphics supplements, or you may wait and cover them in Chapter 14. If you want just a brief sample of applets, you can read only the graphics supplement for this first chapter.

A First Java Application Program

Our first Java program is shown in Listing 1.1. Below the program, we show a sample of the screen output that might be produced when a person runs and interacts with the program. The person who interacts with a program is called the **user.** The text typed in by the user is shown in color. If you run this program—and you should do so—both the text displayed by the program and the text you type will appear in the same color on your computer screen.

A user runs and interacts with a program

The user might or might not be the programmer, that is, the person who wrote the program. As a student, you often are both the programmer and the user, but in a real-world setting, the programmer and user are generally different people. This book is teaching you to be the programmer. One of the first things you need to learn is that you cannot expect the users of your program to know what you want them to do. For that reason, your program must give the user understandable instructions, as we have done in the sample program.

At this point, we just want to give you a feel for the Java language by providing a brief, informal description of the sample program shown in Listing 1.1. *Do not worry if some of the details of the program are not completely clear on this first reading*. This is just a preview of things to come. In Chapter 2, we will explain the details of the Java features used in the program.

The first line

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

tells the compiler that this program uses the class `Scanner`. Recall that for now, we can think of a class as a piece of software that we can use in a program. This class is defined in the package `java.util`, which is short for "Java utility." A **package** is a library of classes that have already been defined for you.

A package is a library of classes

The remaining lines define the class `FirstProgram`, extending from the first open brace ({) to the last close brace (}):

```
public class FirstProgram
{
  . . .
}
```

LISTING 1.1 A Sample Java Program

```
import java.util.Scanner;

public class FirstProgram
{
    public static void main(String[] args)
    {
        System.out.println("Hello out there.");
        System.out.println("I will add two numbers for you.");
        System.out.println("Enter two whole numbers on a line:");

        int n1, n2;


        Scanner keyboard = new Scanner(System.in);
        n1 = keyboard.nextInt();
        n2 = keyboard.nextInt();


        System.out.println("The sum of those two numbers is");
        System.out.println(n1 + n2);
    }
}
```

Gets the `Scanner` *class from the package (library)* `java.util`

Name of the class—your choice. "This program should be in a file named `FirstProgram.java`*"*

Sends output to screen

Says that `n1` *and* `n2` *are variables that hold integers (whole numbers)*

Readies the program for keyboard input

Reads one whole number from the keyboard

Sample Screen Output

```
Hello out there.
I will add two numbers for you.
Enter two whole numbers on a line:
12 30
The sum of those two numbers is
42
```

Within these braces are typically one or more parts called **methods.** Every Java application has a method called `main`, and often other methods. The definition of the method main extends from another open brace to another close brace:

A class contains methods

Every application has a main method

```
public static void main(String[] args)
{
   . . .
}
```

The words `public static void` will have to remain a mystery for now, but they are required. Chapters 5 and 6 will explain these details.

Any **statements,** or instructions, within a method define a task and make up the **body** of the method. The first three statements in our main method's body are the first actions this program performs:

```
System.out.println("Hello out there.");
System.out.println("I will add two numbers for you.");
System.out.println("Enter two whole numbers on a line:");
```

Each of these statements begins with `System.out.println` and causes the quoted characters given within the parentheses to be displayed on the screen on their own line. For example,

```
System.out.println("Hello out there.");
```

causes the line

```
Hello out there.
```

to be written to the screen.

For now, you can consider `System.out.println` to be a funny way of saying "Display what is shown in parentheses." However, we can tell you a little about what is going on here and introduce some terminology. Java programs use things called **software objects** or, more simply, **objects** to perform actions. The actions are defined by methods. `System.out` is an object used to send output to the screen; `println` is the method that performs this action for the object `System.out`. That is, `println` sends what is within its parentheses to the screen. The item or items inside the parentheses are called **arguments** and provide the information the method needs to carry out its action. In each of these first three statements, the argument for the method `println` is a string of characters between quotes. This argument is what `println` writes to the screen.

Objects perform actions when you call its methods

An object performs an action when you **invoke,** or **call,** one of its methods. In a Java program, you write such a **method call,** or **method invocation,** by writing the name of the object, followed by a period—called a **dot** in computer jargon—followed by the method name and some parentheses that might or might not contain arguments.

The next line of the program in Listing 1.1,

Variables store data

```
int n1, n2;
```

says that `n1` and `n2` are the names of variables. A **variable** is something that can store a piece of data. The `int` says that the data must be an integer, that is, a whole number; `int` is an example of a **data type.** A data type specifies a set of possible values and the operations defined for those values. The values of a particular data type are stored in memory in the same format.

A data type specifies a set of values and their operations

The next line

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

enables the program to accept, or **read,** data that a user enters at the keyboard. We will explain this line in detail in Chapter 2.[2]

A program gets, or reads, data from a user

Next, the line

```
n1 = keyboard.nextInt();
```

reads a number that is typed at the keyboard and then stores this number in the variable `n1`. The next line is almost the same except that it reads another number typed at the keyboard and stores this second number in the variable `n2`. Thus, if the user enters the numbers 12 and 30, as shown in the sample output, the variable `n1` will contain the number 12, and the variable `n2` will contain the number 30.

Finally, the statements

```
System.out.println("The sum of those two numbers is");
System.out.println(n1 + n2);
```

display an explanatory phrase and the sum of the numbers stored in the variables `n1` and `n2`. Note that the second line contains the **expression** `n1 + n2` rather than a string of characters in quotes. This expression computes the sum of the numbers stored in the variables `n1` and `n2`. When an output statement like this contains a number or an expression whose value is a number, the number is displayed on the screen. So in the sample output shown in Listing 1.1, these two statements produce the lines

```
The sum of those two numbers is
42
```

Notice that each invocation of `println` displays a separate line of output.

The only thing left to explain in this first program are the semicolons at the end of certain lines. The semicolon acts as ending punctuation, like a period in an English sentence. A semicolon ends an instruction to the computer.

Of course, Java has precise rules for how you write each part of a program. These rules form the **grammar** for the Java language, just as the rules for the English language make up its grammar. However, Java's rules are more precise. The grammatical rules for any language, be it a programming language or a natural language, are called the **syntax** of the language.

Syntax is the set of grammatical rules for a language

[2] As you will see in the next chapter, you can use some other name in place of keyboard, but that need not concern us now. Anyway, keyboard is a good word to use here.

RECAP Invoking (Calling) a Method

A Java program uses objects to perform actions that are defined by methods. An object performs an action when you invoke, or call, one of its methods. You indicate this in a program by writing the object name, followed by a period—called a dot—then the method name, and finally a pair of parentheses that can contain arguments. The arguments are information for the method.

EXAMPLES:

```
System.out.println("Hello out there.");
n1 = keyboard.nextInt();
```

In the first example, `System.out` is the object, `println` is the method, and `"Hello out there."` is the argument. When a method requires more than one argument, you separate the arguments with commas. A method invocation is typically followed by a semicolon.

In the second example, `keyboard` is the object and `nextInt` is the method. This method has no arguments, but the parentheses are required nonetheless.

FAQ Why do we need an `import` for input but not for output?

The program in Listing 1.1 needs the line

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

to enable keyboard input, such as the following:

```
n1 = keyboard.nextInt();
```

Why don't we need a similar import to enable screen `output` such as

```
System.out.println("Hello out there.");
```

The answer is rather dull. The package that includes definitions and code for screen output is imported automatically into a Java program.

SELF-TEST QUESTIONS

11. What would the following statement, when used in a Java program, display on the screen?

```
System.out.println("Java is great!");
```

12. Write a statement or statements that can be used in a Java program to display the following on the screen:

```
Java for one.
Java for all.
```

13. Suppose that mary is an object that has the method `increaseAge`. This method takes one argument, an integer. Write an invocation of the method `increaseAge` by the object mary, using the argument 5.

14. What is the meaning of the following line in the program in Listing 1.1?

```
n1 = keyboard.nextInt();
```

15. Write a complete Java program that uses `System.out.println` to display the following to the screen when the program is run:

```
Hello World!
```

Your program does nothing else. Note that you do not need to fully understand all the details of the program in order to write it. You can simply follow the model of the program in Listing 1.1. (You do want to understand all the details eventually, but that may take a few more chapters.)

Writing, Compiling, and Running a Java Program

Writing a Java program

A Java program is divided into smaller parts called classes. Each program can consist of any number of class definitions. Although we wrote only one class—`FirstProgram`—for the program in Listing 1.1, in fact, the program uses two other classes: `System` and `Scanner`. However, these two classes are provided for you by Java.

Each class is in a file whose name ends in `.java`

You can write a Java class by using a simple text editor. For example, you could use Notepad in a Windows environment or TextEdit on a Macintosh system. Normally, each class definition you write is in a separate file. Moreover, the name of that file must be the name of the class, with `.java` added to the end. For example, the class `FirstProgram` must be in a file named `FirstProgram.java`.

Before you can run a Java program, you must translate its classes into a language that the computer can understand. As you saw earlier in this chapter, this translation process is called compiling. As a rule, you do not need to compile classes like `Scanner` that are provided for you as part of Java. You normally need compile only the classes that you yourself write.

Compiling a Java program

To compile a Java class using the free Java system distributed by Oracle® for Windows, Linux, or Solaris, you use the command `javac` followed by the name of the file containing the class. For example, to compile a class named `MyClass` that is in a file named `MyClass.java`, you give the following command to the operating system:

Use the command `javac` to compile

```
javac MyClass.java
```

Thus, to compile the class in Listing 1.1, you would give the following command:

```
javac FirstProgram.java
```

Bytecode is in a file whose name ends in `.class`

When you compile a Java class, the translated version of the class—its bytecode—is placed in a file whose name is the name of the class followed by `.class`. So when you compile a class named `MyClass` in the file `MyClass.java`, the resulting bytecode is stored in a file named `MyClass.class`. When you compile the file named `FirstProgram.java`, the resulting bytecode is stored in a file named `FirstProgram.class`.

VideoNote
Compiling a Java program

Although a Java program can involve any number of classes, you run only the class that you think of as the program. This class will contain a `main` method beginning with words identical to or very similar to

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

These words will likely, but not always, be someplace near the beginning of the file. The critical words to look for are `public static void main`. The remaining portion of the line might use somewhat different wording.

Use the command `java` to execute

You run a Java program by giving the command `java`, followed by the name of the class you think of as the program. For example, to run the program in Listing 1.1, you would give the following one-line command:

```
java FirstProgram
```

Note that you write the class name, such as `FirstProgram`, not the name of the file containing the class or its bytecode. That is, you omit any `.java` or `.class` ending. When you run a Java program, you are actually running the Java bytecode interpreter on the compiled version of your program.

The easiest way to write, compile, and run a Java program is to use an **integrated development environment**, or **IDE.** An IDE combines a text editor with menu commands for compiling and running a Java program. IDEs such as BlueJ, Eclipse, and NetBeans are free and available for Windows, Mac OS, and other systems. Appendix 1 provides links to these IDEs and other resources for writing Java programs.

FAQ I tried to run the sample program in Listing 1.1. After I typed two numbers on a line, nothing happened. Why?

When you type a line of data at the keyboard for a program to read, you will see the characters you type, but the Java program does not actually read your data until you press the Enter (Return) key. Always press the Enter key when you have finished typing a line of input data at the keyboard.

SELF-TEST QUESTIONS

16. Suppose you define a class named `YourClass` in a file. What name should the file have?

17. Suppose you compile the class `YourClass`. What will be the name of the file containing the resulting bytecode?

1.3 PROGRAMMING BASICS

'The time has come,' the Walrus said,
'To talk of many things:
Of shoes–and ships–and sealing wax–
Of cabbages–and kings . . .'

—LEWIS CARROLL, *Through the Looking-Glass*

Programming is a creative process. We cannot tell you exactly how to write a program to do whatever task you might want it to perform. However, we can give you some techniques that experienced programmers have found to be extremely helpful. In this section, we discuss some basics of these techniques. They apply to programming in almost any programming language and are not particular to Java.

Object-Oriented Programming

Software objects act and interact

Java is an **object-oriented programming** language, abbreviated **OOP.** What is OOP? The world around us is made up of objects, such as people, automobiles, buildings, trees, shoes, ships, sealing wax, cabbages, and kings. Each of these objects has the ability to perform certain actions, and each action can affect some of the other objects in the world. OOP is a programming methodology that views a program as similarly consisting of objects that can act alone or interact with one another. An object in a program—that is, a software object—might represent a real-world object, or it might be an abstraction.

For example, consider a program that simulates a highway interchange so that traffic flow can be analyzed. The program would have an object to represent each automobile that enters the interchange, and perhaps other objects to simulate each lane of the highway, the traffic lights, and so on. The interactions among these objects can lead to a conclusion about the design of the interchange.

The values of an object's attributes define its state

Object-oriented programming comes with its own terminology. An object has characteristics, or **attributes.** For example, an automobile object might have attributes such as its name, its current speed, and its fuel level. The values of an object's attributes give the object a **state.** The actions that an object can take are called **behaviors.** As we saw earlier, each behavior is defined by a piece of Java code called a method.

A class is a blueprint for objects

Objects of the same kind are said to have the same data type and belong to the same class. A **class** defines a kind of object; it is a blueprint for creating objects. The data type of an object is the name of its class. For example, in a highway simulation program, all the simulated automobiles might belong to the same class—probably called Automobile—and so their data type is Automobile.

All objects of a class have the same attributes and behaviors. Thus, in a simulation program, all automobiles have the same behaviors, such as moving forward and moving backward. This does not mean that all simulated automobiles are identical. Although they have the same attributes, they can have different states. That is, a particular attribute can have different values among the automobiles. So we might have three automobiles having different makes and traveling at different speeds. All this will become clearer when we begin to write Java classes.

As you will see, this same object-oriented methodology can be applied to any sort of computer program and is not limited to simulation programs. Object-oriented programming is not new, but its use in applications outside of simulation programs did not become popular until the early 1990s.

RECAP Objects, Methods, and Classes

An object is a program construction that has data—called attributes—associated with it and that can perform certain actions known as behaviors. A class defines a type or kind of object. It is a blueprint for defining the objects. All objects of the same class have the same kinds of data and the same behaviors. When the program is run, each object can act alone or interact with other objects to accomplish the program's purpose. The actions performed by objects are defined by methods.

FAQ What if I know some other programming language?

If Java is your first programming language, you can skip the answer to this question. If you know some other programming language, the discussion here may help you to understand objects in terms of things you already know about. If that other programming language is object oriented, such as C++, C#, Python, or Ruby, you have a good idea of what objects, methods, and classes are. They are basically the same in all object-oriented programming languages, although some other languages might use another term to mean the same thing as *method*. If your familiarity is with an older programming language that does not use objects and classes, you can think of objects in terms of other, older programming constructs. For example, if you know about variables and functions or procedures, you can think of an object as a variable that has multiple pieces of data and its own functions or procedures. Methods are really the same thing as what are called *functions* or *procedures* in older programming languages.

Object-oriented programming uses classes and objects, but it does not use them in just any old way. There are certain design principles that must be followed. The following are three of the main design principles of object-oriented programming:

OOP design principles

Encapsulation
Polymorphism
Inheritance

Encapsulation sounds as though it means putting things into a capsule or, to say it another way, packaging things up. This intuition is basically correct. The most important part of encapsulation, however, is not simply that things are put into a capsule, but that only part of what is in the capsule is visible. When you produce a piece of software, you should describe it in a way that tells other programmers how to use it, but that omits all the details of how the software works. Note that encapsulation hides the fine detail of what is inside the "capsule." For this reason, encapsulation is often called **information hiding.**

Encapsulation packages and hides detail

The principles of encapsulation apply to programming in general, not just to object-oriented programming. But object-oriented languages enable a programmer not only to realize these principles but also to enforce them. Chapter 5 will develop the concept of encapsulation further.

Polymorphism comes from a Greek word meaning "many forms." The basic idea of polymorphism is that it allows the same program instruction to mean different things in different contexts. Polymorphism commonly occurs in English, and its use in a programming language makes the programming language more like a human language. For example, the English instruction "Go play your favorite sport" means different things to different people. To one person, it means to play baseball. To another person, it means to play soccer.

Polymorphism also occurs in everyday tasks.[3] Imagine a person who whistles for her pets to come to dinner. Her dog runs, her bird flies, and her fish swim to the top of their tank. They all respond in their own way. The come-to-dinner whistle doesn't tell the animals how to come to dinner, just to come. Likewise when you press the "on" button on your laptop, your iPod, or your toothbrush, each of them responds appropriately. In a programming language such as Java, polymorphism means that one method name, used as an instruction, can cause different actions, depending on the kinds of objects that perform the action. For example, a method named showOutput might display the data in an object. But the number of data items it displays and their format depend on the kind of object that carries out the action. We will explain polymorphism more fully in Chapter 8.

Polymorphism enables objects to behave appropriately

Inheritance organizes related classes

Inheritance is a way of organizing classes. You can define common attributes and behaviors once and have them apply to a whole collection of classes. By defining a general class, you can use inheritance later to define specialized classes that add to or revise the details of the general class.

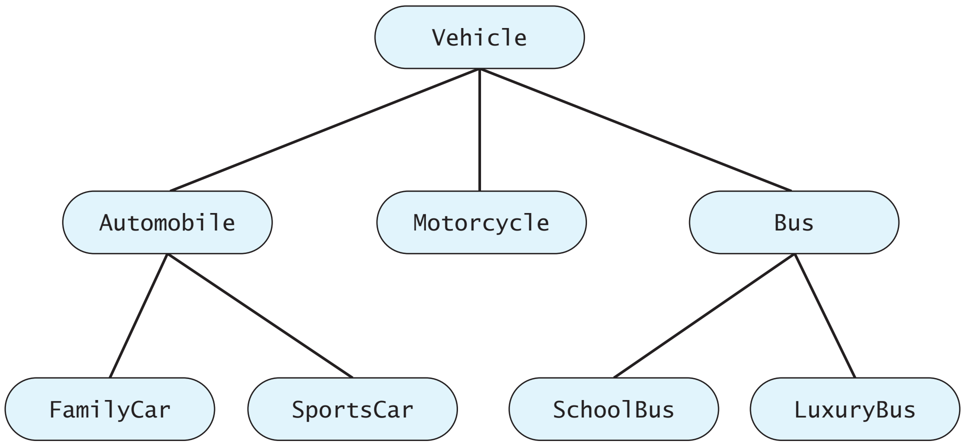
An example of such a collection of classes is shown in Figure 1.4. At each level, the classifications become more specialized. The class `Vehicle` has certain properties, like possessing wheels. The classes `Automobile`, `Motorcycle`, and `Bus` "inherit" the property of having wheels, but add more properties or restrictions. For example, an `Automobile` object has four wheels, a `Motorcycle` object has two wheels, and a Bus object has at least four wheels. Inheritance enables the programmer to avoid the repetition of programming instructions for each class. For example, everything that is true of every object of type `Vehicle`, such as "has wheels," is described only once, and it is inherited by the classes `Automobile`, `Motorcycle`, and `Bus`. Without inheritance, each of the classes `Automobile`, `Motorcycle`, `Bus`, `SchoolBus`, `LuxuryBus`, and so forth would have to repeat descriptions such as "has wheels." Chapter 8 will explain inheritance more fully.

RECAP Object-Oriented Programming

Object-oriented programming, or OOP, is a programming methodology that defines objects whose behaviors and interactions accomplish a given task. OOP follows the design principles of encapsulation, polymorphism, and inheritance.

[3] The examples here are based on those by Carl Alphonce in "Pedagogy and Practice of Design Patterns and Objects First: A One-Act Play." *ACM SIGPLAN Notices* 39, 5 (May 2004), 7–14.

FIGURE 1.4 An Inheritance Hierarchy



Algorithms

Objects have behaviors that are defined by methods. You as a programmer need to design these methods by giving instructions for carrying out the actions. The hardest part of designing a method is not figuring out how to express your solution in a programming language. The hardest part is coming up with a plan or strategy for carrying out the action. This strategy is often expressed as something called an algorithm.

An algorithm is like a recipe

An **algorithm** is a set of directions for solving a problem. To qualify as an algorithm, the directions must be expressed so completely and so precisely that somebody can follow them without having to fill in any details or make any decisions that are not fully specified in the instructions. An algorithm can be written in English, a programming language such as Java, or in **pseudocode,** which is a combination of English and a programming language.

Algorithms are often written in pseudocode

An example may help to clarify the notion of an algorithm. Our first sample algorithm finds the total cost of a list of items. For example, the list of items might be a shopping list that includes the price of each item. The algorithm would then compute the total cost of all the items on the list. The algorithm is as follows:

Algorithm to compute the total cost of a list of items

VideoNote
Writing an algorithm

1. Write the number 0 on the blackboard.
2. Do the following for each item on the list:
   - Add the cost of the item to the number on the blackboard.
   - Replace the old number on the blackboard with the result of this addition.
3. Announce that the answer is the number written on the blackboard.

Most algorithms need to store some intermediate results. This algorithm uses a blackboard to store intermediate results. If the algorithm is written in the Java language and run on a computer, intermediate results are stored in the computer's memory.

RECAP Algorithm

An algorithm is a set of directions for solving a problem. To qualify as an algorithm, the directions must be expressed completely and precisely.

RECAP Pseudocode

Pseudocode is a mixture of English and Java. When using pseudocode, you simply write each part of the algorithm in whatever language is easiest for you. If a part is easier to express in English, you use English. If another part is easier to express in Java, you use Java.

SELF-TEST QUESTIONS

18. What is a method?

19. What is the relationship between classes and objects?

20. Do all objects of the same class have the same methods?

21. What is encapsulation?

22. What is information hiding?

23. What is polymorphism?

24. What is inheritance?

25. What is an algorithm?

26. What is pseudocode?

27. What attributes would you want for an object that represents a song?

28. Write an algorithm that counts the number of values that are odd in a list of integers.

Testing and Debugging

The best way to write a correct program is to carefully design the necessary objects and the algorithms for the objects' methods. Then you carefully translate everything into a programming language such as Java. In other words, the best way to eliminate errors is to avoid them in the first place. However, no matter how carefully you proceed, your program might still contain some errors. When you finish writing a program, you should test it to see whether it performs correctly and then fix any errors you find.

A mistake in a program is called a **bug.** For this reason, the process of eliminating mistakes in your program is called **debugging.** There are three commonly recognized kinds of bugs or errors: syntax errors, run-time errors, and logic errors. Let's consider them in that order.

Syntax errors are grammatical mistakes

A **syntax error** is a grammatical mistake in your program. You must follow very strict grammatical rules when you write a program. Violating one of these rules—for example, omitting a required punctuation mark—is a syntax error. The compiler will detect syntax errors and provide an error message indicating what it thinks the error is. If the compiler says you have a syntax error, you probably do. However, the compiler is only guessing at what the error is, so it could be incorrect in its diagnosis of the problem.

RECAP Syntax

The syntax of a programming language is the set of grammatical rules for the language—that is, the rules for the correct way to write a program or part of a program. The compiler will detect syntax errors in your program and provide its best guess as to what is wrong.

An error that is detected when your program is run is called a **run-time error.** Such an error will produce an error message. For example, you might accidentally try to divide a number by zero. The error message might not be easy to understand, but at least you will know that something is wrong. Sometimes the error message can even tell you exactly what the problem is.

Run-time errors occur during execution

If the underlying algorithm for your program contains a mistake, or if you write something in Java that is syntactically correct but logically wrong, your program could compile and run without any error message. You will have written a valid Java program, but you will not have written the program you wanted. The program will run and produce output, but the output will be incorrect. In this case, your program contains a **logic error.** For example, if you were to mistakenly use a plus sign instead of a minus sign, you would make a logic error. You could compile and run your program with no error messages, but the program would give the wrong output. Sometimes a logic error will lead to a run-time error that produces an error message. But often a logic error will not give you any error messages. For this reason, logic errors are the hardest kind of error to locate.

Logic errors are conceptual mistakes in the program or algorithm

GOTCHA Coping with "Gotchas"

Don't let a gotcha get you

Any programming language has details that can trip you up in ways that are surprising or hard to deal with. These sorts of problems are often called pitfalls, but a more colorful term is *gotchas*. A gotcha is like a trap waiting to catch you. When you get caught in the trap, the trap has "got you" or, as it is more commonly pronounced, "gotcha."

In this book, we have "Gotcha" sections like this one that warn you about many of the most common pitfalls and tell you how to avoid them or cope with them. ■

GOTCHA Hidden Errors

VideoNote
Recognizing a hidden error

Just because your program compiles and runs without any errors and even produces reasonable-looking output does not mean that your program is correct. You should always run your program with some test data that gives predictable output. To do this, choose some data for which you can compute the correct results, either by using pencil and paper, by looking up the answer, or by some other means. Even this testing does not guarantee that your program is correct, but the more testing you do, the more confidence you can have in your program. ■

SELF-TEST QUESTIONS

29. What is a syntax error?

30. What is a logic error?

31. What kinds of errors are likely to produce error messages that will alert you to the fact that your program contains an error?

32. Suppose you write a program that is supposed to compute the day of the week (Sunday, Monday, and so forth) on which a given date (like December 1, 2014) will fall. Now suppose that you forget to account for leap years. Your program will then contain an error. What kind of program error is it?

Software Reuse

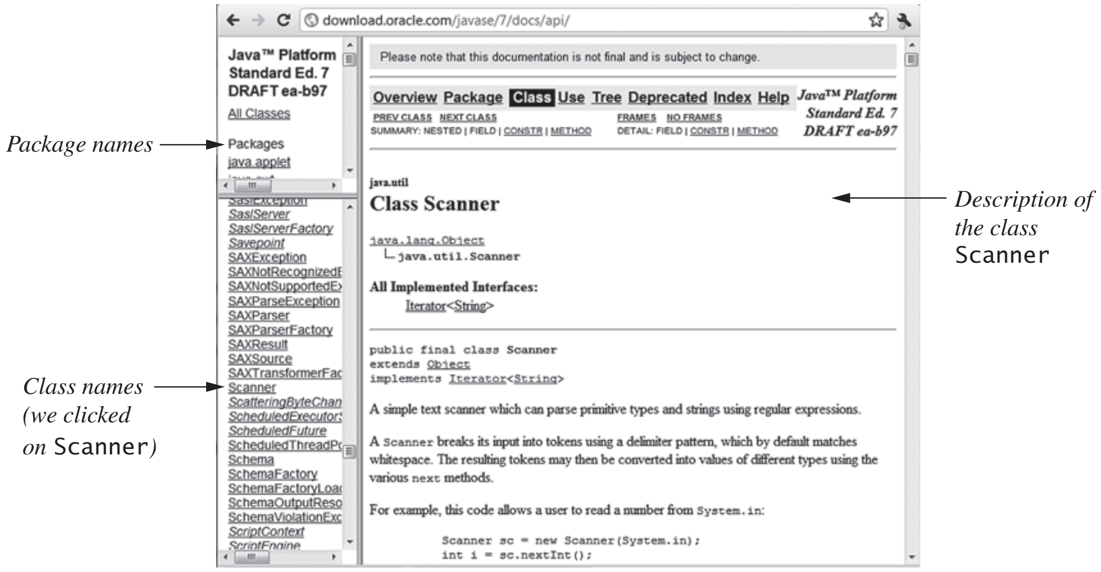
When you first start to write programs, you can easily get the impression that you must create each program entirely from scratch. However, typical software is not produced this way. Most programs contain some components that already exist. Using such components saves time and money. Furthermore, existing components have probably been used many times, so they likely are better tested and more reliable than newly created software.

For example, a highway simulation program might include a new highway object to model a new highway design but would probably model automobiles by using an automobile class that was already designed for some other program. To ensure that the classes you use in your programs are easily reusable, you must design them to be reusable. You must specify exactly how objects of that class interact with other objects. This is the principle of encapsulation that we mentioned earlier. But encapsulation is not the only principle you must follow. You must also design your class so that the objects are general and not specific to one particular program. For example, if your program requires that all simulated automobiles move only forward, you should still include a reverse in your automobile class, because some other simulation may require automobiles to back up. We will return to the topic of reusability after we learn more details about the Java language and have some examples to work with.

Java provides a library of classes for you

Besides reusing your own classes, you can and will use classes that Java provides. For example, we have already used the standard classes `Scanner` and `System` to perform input and output. Java comes with a collection of many classes known as the **Java Class Library,** sometimes called the **Java Application Programming Interface,** or **API.** The classes in this collection are organized into packages. As you saw earlier, the class Scanner, for example, is in the package `java.util`. From time to time we will mention or use classes within the Java Class Library. You should become familiar with the documentation provided for the Java Class Library on the Oracle® Web site. At this writing, the link to this documentation is http://docs.oracle.com/javase/7/docs/api/. Figure 1.5 gives an example of this documentation.

FIGURE 1.5 The Documentation for the Class Scanner



Please note that this documentation is not final and is subject to change.

java.util

Class Scanner

```
java.lang.Object
  └ java.util.Scanner
```

All Implemented Interfaces:

Iterator<String>

```
public final class Scanner
extends Object
implements Iterator<String>
```

A simple text scanner which can parse primitive types and strings using regular expressions.

A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various `next` methods.

For example, this code allows a user to read a number from `System.in`:

```
        Scanner sc = new Scanner(System.in);
        int i = sc.nextInt();
```

1.4 GRAPHICS SUPPLEMENT

Have a nice day.

—COMMON FAREWELL

Each of Chapters 1 through 10 has a graphics section like this one that describes how to write programs that include various kinds of graphics displays. We typically will display the graphics inside of an applet because it is easier to do, especially for beginners. However, sometimes we will use a windowing interface within an application program to display the graphics. Chapter 2 will introduce you to this approach.

Since some people prefer to delay coverage of graphics until after a programmer, such as yourself, has mastered the more elementary material, you may skip these supplements without affecting your understanding of the rest of the book. In order to cover graphics this early, we will have to resort to some "magic formulas"—that is, code that we will tell you how to use but not fully explain until later in the book. These graphics supplements do build on each other. If you want to cover the graphics supplement in one chapter, you will need to first read all or most of the graphics supplements in previous chapters.

The material on applets and graphics presented here uses classes, objects, and methods. You know that objects are entities that store data and can take actions. In this section, we will use objects only for taking actions, and we will use only one kind of object. Our objects will usually be named canvas and will have various methods that can draw figures—such as ovals—inside an applet display.

REMEMBER You Can Display Graphics in Applets and Application Programs

Whether you write an applet or an application program to display graphics depends on your objective. You would write an applet if you want to have a graphical feature on a Web page. Otherwise, you would write an application program.

A Sample Graphics Applet

Listing 1.2 contains an applet that draws a happy face. Let's examine the code by going through it line by line.

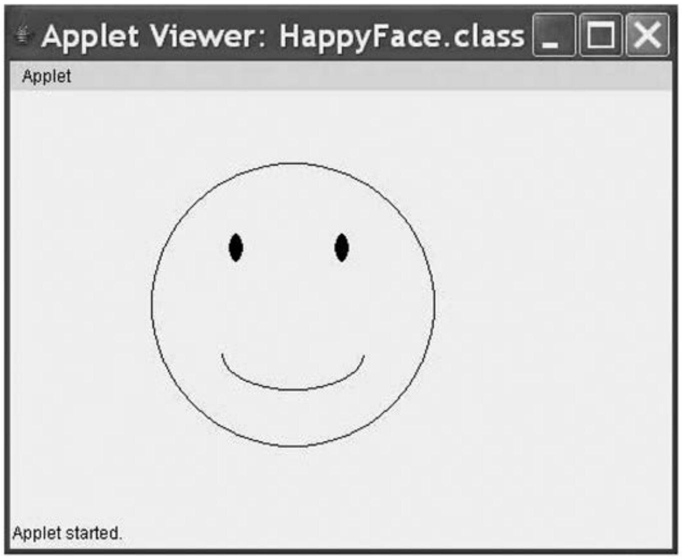
The line

```
import javax.swing.JApplet;
```

LISTING 1.2 Drawing a Happy Face

```
import javax.swing.JApplet;
import java.awt.Graphics;
public class HappyFace extends JApplet
{
   public void paint(Graphics canvas)
   {
      super.paint(canvas);
      canvas.drawOval(100, 50, 200, 200);
      canvas.fillOval(155, 100, 10, 20);
      canvas.fillOval(230, 100, 10, 20);
      canvas.drawArc(150, 160, 100, 50, 180, 180);
   }
}
```

Applet Output



says that this applet—like all applets—uses the class `JApplet` that is in the Swing library (package). The line

Applets use the packages Swing and AWT

```
import java.awt.Graphics;
```

says that this applet also uses the class Graphics from the AWT library (package). Applets often use classes in the AWT library in addition to classes in the Swing library.

The next line

```
public class HappyFace extends JApplet
```

begins the class definition for the applet. It is named `HappyFace`. The words `extends JApplet` indicate that we are defining an applet, as opposed to some other kind of class. Although you need not worry about further details yet, we are using inheritance to create the class `HappyFace` based upon an existing class `JApplet`.

The applet contains one method—paint—whose definition begins with

```
public void paint(Graphics canvas)
```

An applet's paint method draws its graphics

The `paint` method specifies what graphics are drawn in the applet. Each of the four statements within the body of the method is an instruction to draw a figure. The paint method is invoked automatically when the applet is run.

We will not discuss the details of method definitions until Chapter 5, but we will tell you enough here to allow you to define the method `paint` to do some simple graphics. The method invocation

```
super.paint(canvas);
```

tells Java to apply the default drawing operations to this applet. If this is left out the window may not be drawn correctly. The method invocation

```
canvas.drawOval(100, 50, 200, 200);
```

draws the big circle that forms the outline of the face. The first two numbers tell where on the screen the circle is drawn. The method `drawOval`, as you may have guessed, draws ovals. The last two numbers give the width and height of the oval. To obtain a circle, you make the width and height the same size, as we have done here. The units used for these numbers are called *pixels*, and we will describe them shortly.

The two method invocations

```
canvas.fillOval(155, 100, 10, 20);
canvas.fillOval(230, 100, 10, 20);
```

draw the two eyes. The eyes are "real" ovals that are taller than they are wide. Also notice that the method is called `fillOval`, not `drawOval`, which means it draws an oval that is filled in.

The last invocation

```
canvas.drawArc(150, 160, 100, 50, 180, 180);
```

draws the mouth. We will explain the meaning of all of these arguments next.

Size and Position of Figures

All measurements within a screen display are given not in inches or centimeters but in pixels. A **pixel**—short for picture element—is the smallest length your screen is capable of showing. A pixel is not an absolute unit of length like an inch or a centimeter. The size of a pixel can be different on different screens, but it will always be a small unit. You can think of your computer screen as being covered by small squares, each of which can be any color. You cannot show anything smaller than one of these squares. A pixel is one of these squares, but when used as measure of length, a pixel is the length of the side of one of these squares.[4] If you have shopped for a digital camera, you have undoubtedly heard the term *pixel* or *megapixel*. The meaning of the word *pixel* when used in Java applets is the same as its meaning when describing pictures from a digital camera. A **megapixel** is just a million pixels.

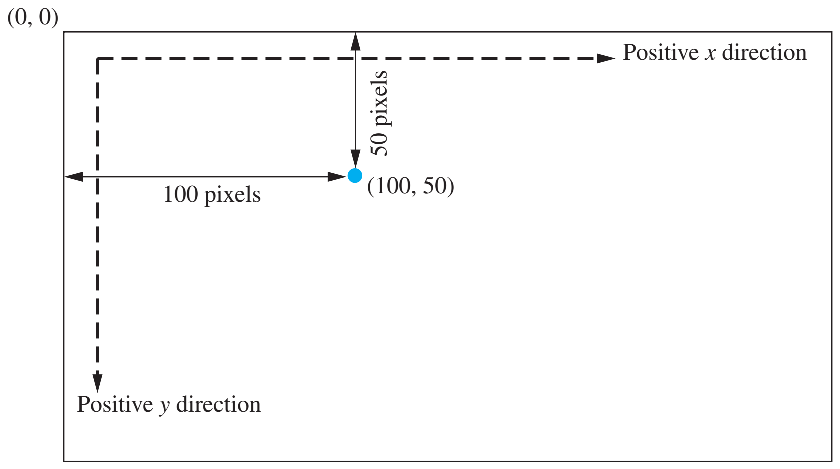
A pixel is the smallest length shown on a screen

Figure 1.6 shows the **coordinate system** used to position figures inside of an applet or other kind of Java window-like display. Think of the large rectangle as outlining the drawing area that is displayed on the screen. The coordinate system assigns two numbers to each point inside the rectangle. The numbers are known as the ***x*-coordinate** and the **y-coordinate** of the point. The ***x*-coordinate** is the number of pixels from the left edge of the rectangle to the point. The ***y*-coordinate** is the number of pixels from the top edge of the rectangle to the point. The coordinates are usually written within parentheses and separated by a comma, with the *x*-coordinate first. So the point marked with a blue dot in Figure 1.6 has the coordinates (100, 50); 100 is the *x*-coordinate and 50 is the *y*-coordinate.

A coordinate system positions points on the screen

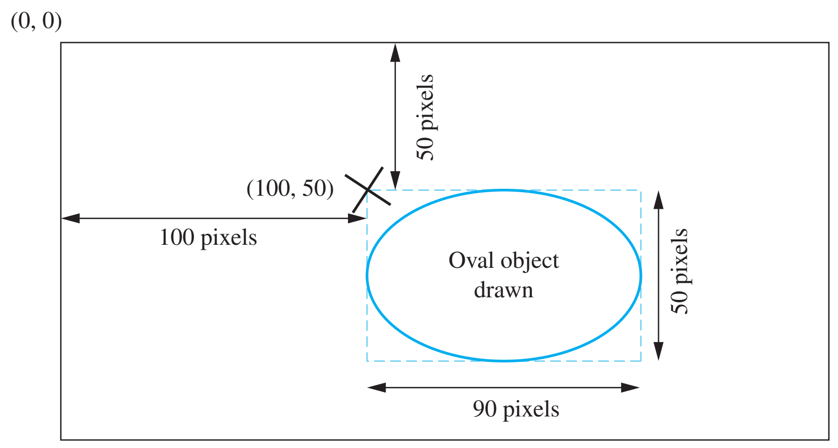
Each coordinate in this system is greater than or equal to zero. The *x*-coordinate gets larger as you go to the right from point (0, 0). The *y*-coordinate gets larger as you go down from point (0, 0). If you have studied *x*- and *y*-coordinates in a math class, these are the same, with one change. In other coordinate systems, the *y*-coordinates increase as they go *up* from point (0, 0).

FIGURE 1.6 Screen Coordinate System



[4] Strictly speaking, a pixel need not be a square but could be rectangular. However, we do not need to go into such fine detail here.

FIGURE 1.7 The Oval Drawn by canvas.drawOval (100, 50, 90, 50)



You position a rectangle in this graphical coordinate system at coordinates (*x*, *y*) by placing its upper left corner at the point (*x*, *y*). For example, the rectangle given by the dashed blue lines in Figure 1.7 is positioned at point (100, 50), which is marked with a black X. You position a figure that is not a rectangle at point (*x*, *y*) by first enclosing it in an imaginary rectangle that is as small as possible but still contains the figure and then by placing the upper left corner of this enclosing rectangle at (*x*, *y*). For example, in Figure 1.7 the oval is also positioned at point (100, 50). If the applet contains only an oval and no rectangle, only the oval shows on the screen. But an imaginary rectangle is still used to position the oval.

Drawing Ovals and Circles

The oval in Figure 1.7 is drawn by the Java statement

```
canvas.drawOval(100, 50, 90, 50);
```

The first two numbers are the *x*- and *y*-coordinates of the upper left corner of the imaginary rectangle that encloses the oval. That is, these two numbers are the coordinates of the position of the figure drawn by this statement. The next two numbers give the width and height of the rectangle containing the oval (and thus the width and height of the oval itself). If the width and height are equal, you get a circle.

Now let's return to the statements in the body of the method `paint`:

`drawOval` and `fillOval` draw ovals or circles

```
canvas.drawOval(100, 50, 200, 200);
canvas.fillOval(155, 100, 10, 20);
canvas.fillOval(230, 100, 10, 20);
```

In each case, the first two numbers are the *x*- and *y*-coordinates of the upper left corner of an imaginary rectangle that encloses the figure being drawn. The first statement draws the outline of the face at position (100, 50). Since the width and height—as given by the last two arguments—have the same value, 200, we get a circle whose diameter is 200. The next two statements draw filled ovals for the eyes positioned at the points (155, 100) and (230, 100). The eyes are each 10 pixels wide and 20 pixels high. The results are shown in Listing 1.2.

RECAP The Methods drawOval and fillOval

SYNTAX

```
canvas.drawOval(x, y, Width, Height);
canvas.fillOval(x, y, Width, Height);
```

The method `drawOval` draws the outline of an oval that is *Width* pixels wide and *Height* pixels high. The oval is placed so that the upper left corner of a tightly enclosing rectangle is at the point (*x*, *y*).

The method `fillOval` draws the same oval as `drawOval` but fills it in.

Drawing Arcs

Arcs, such as the smile on the happy face in Listing 1.2, are specified as a portion of an oval. For example, the following statement from Listing 1.2 draws the smile on the happy face:

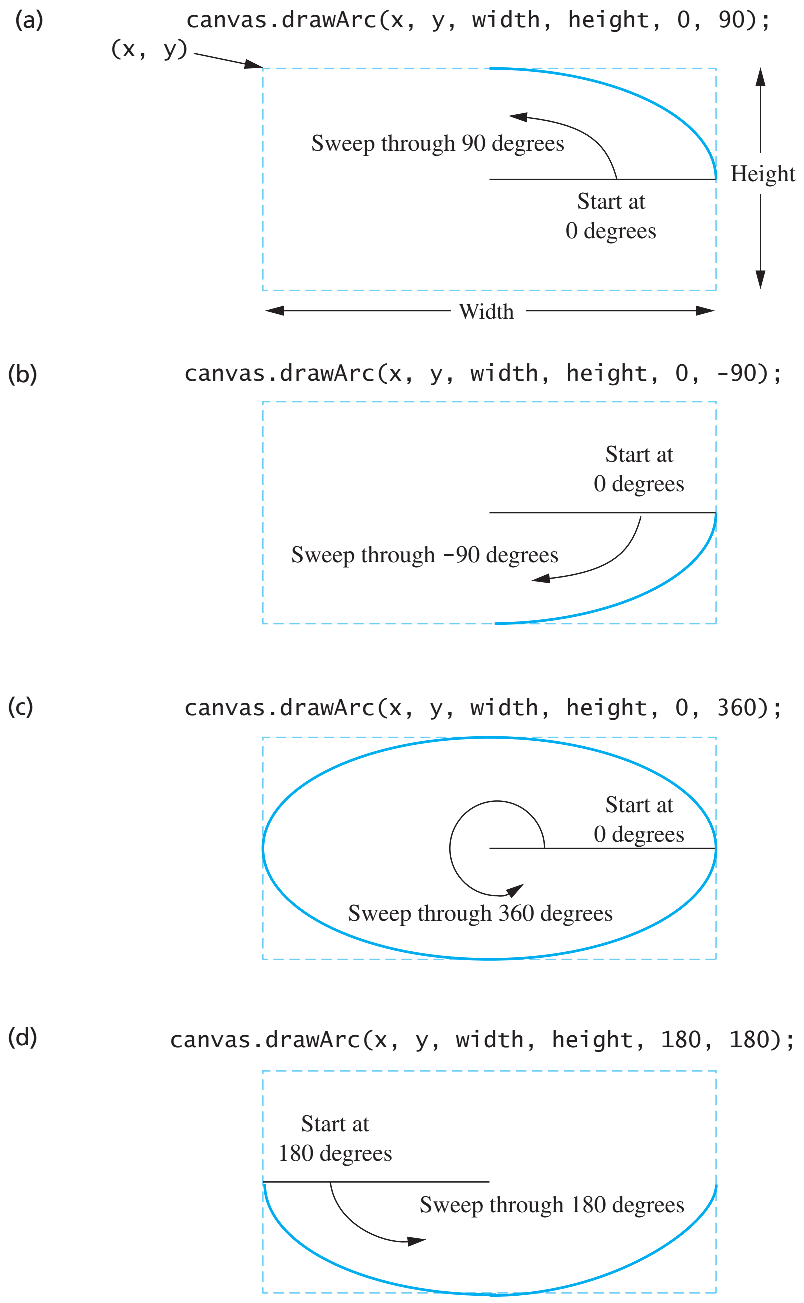
```
canvas.drawArc(150, 160, 100, 50, 180, 180);
```

The first two arguments give the position of an invisible rectangle. The upper left corner of this rectangle is at the point (150, 160). The next two arguments specify the size of the rectangle; it has width 100 and height 50. Inside this invisible rectangle, imagine an invisible oval with the same width and height as the invisible rectangle. The last two arguments specify the portion of this invisible oval that is made visible. In this example, the bottom half of the oval is visible and forms the smile. Let's examine these last two arguments more closely.

drawArc draws part of an oval

The next-to-last argument of `drawArc` specifies a start angle in degrees. The last argument specifies how many degrees of the oval's arc will be made visible. The rightmost end of the oval's horizontal equator is at zero degrees. As you move along the oval's edge in a counterclockwise direction, the degrees increase in value. For example, Figure 1.8a shows a start angle of 0 degrees; we measure 90 degrees along the oval in a counterclockwise direction, making one quarter of the oval visible. Conversely, as you move along the oval in a clockwise direction, the degrees decrease in value. For example, in Figure 1.8b, we start at 0 and move –90 degrees in a clockwise direction, making a different quarter of the oval visible. If the last argument is 360, you move counterclockwise through 360 degrees, making the entire oval visible, as Figure 1.8c shows.

FIGURE 1.8 Specifying an Arc



(a)

```
canvas.drawArc(x, y, width, height, 0, 90);
```

(b)

```
canvas.drawArc(x, y, width, height, 0, -90);
```

(c)

```
canvas.drawArc(x, y, width, height, 0, 360);
```

(d)

```
canvas.drawArc(x, y, width, height, 180, 180);
```

RECAP drawArc

SYNTAX

```
canvas.drawArc(x, y, Width, Height, StartAngle, ArcAngle);
```

Draws an arc that is part of an oval placed so the upper left corner of a tightly enclosing rectangle is at the point (*x*, *y*). The oval's width and height are *Width* and *Height,* both in pixels. The portion of the arc drawn is given by *StartAngle* and *ArcAngle,* both given in degrees. The rightmost end of the oval's horizontal equator is at 0 degrees. You measure positive angles in a counterclockwise direction and negative angles in a clockwise direction. Beginning at *StartAngle,* you measure *ArcAngle* degrees along the oval to form the arc. Figure 1.8 gives some examples of arcs.

FAQ What is canvas?

The identifier canvas names an object that does the drawing. Note that canvas is a "dummy variable" that stands for an object that Java supplies to do the drawing. You need not use the identifier canvas, but you do need to be consistent. If you change one occurrence of canvas to, say, pen, you must change all occurrences of canvas to pen. Thus, the method paint shown in Listing 1.2 could be written as follows:

```
public void paint (Graphics pen)
{
    super.paint(pen)
    pen.drawOval(100, 50, 200, 200);
    pen.fillOval(155, 100, 10, 20);
    pen.fillOval(230, 100, 10, 20);
    pen.drawArc(150, 160, 100, 50, 180, 180);
}
```

This definition and the one given in Listing 1.2 are equivalent.

Finally, Figure 1.8d illustrates an arc that begins at 180 degrees, so it starts on the left end of the invisible oval. The last argument is also 180, so the arc is made visible through 180 degrees in the counterclockwise direction, or halfway around the oval. The smile on the happy face in Listing 1.2 uses this same arc.

Running an Applet

You compile an applet in the same way that you compile any other Java class. However, you run an applet differently from other Java programs. The normal way to run an applet is as part of a Web page. The applet is then viewed through a Web browser. We will discuss this means of viewing an applet in Chapter 14 (on the book's Web site).

An applet viewer will run an applet

You need not know how to embed an applet in a Web page to run it, however. Instead, you can use an **applet viewer,** a program designed to run applets as stand-alone programs. The easiest way to do this is to run the applet from an integrated development environment (IDE), such as the ones mentioned earlier in this chapter. Every IDE has a menu command such as Run Applet, Run, Execute, or something similar. Appendix 2 explains how to use Oracle's applet viewer.

Ending an applet

The way to end an applet depends on how you are running it. If you are using an IDE or other applet viewer, you end the applet display by clicking the close-window button with your mouse. The close-window button will likely be as shown in Listing 1.2, but it might have a different location or appearance, depending on your computer and operating system. In that case, the close-window button will probably be like those on other windows on your computer. If you are running the applet from a Web site, the applet stays until you close or navigate away from the page it is on.

SELF-TEST QUESTIONS

VideoNote
Another applet example

33. How would you change the applet program in Listing 1.2 so that the eyes are circles instead of ovals?

34. How would you change the applet program in Listing 1.2 so that the face frowns? (*Hint:* Turn the smile upside down by changing the arguments in the call to the method `drawArc`.)

CHAPTER SUMMARY

- A computer's main memory holds the program that is currently executing, and it also holds many of the data items that the program is manipulating. A computer's main memory is divided into a series of numbered locations called bytes. This memory is volatile: The data it holds disappears when the computer's power is off.

- A computer's auxiliary memory is used to hold data in a more or less permanent way. Its data remains even when the computer's power is off. Hard disk drives, flash drives, CDs, and DVDs are examples of auxiliary memory.

- A compiler is a program that translates a program written in a high-level language like Java into a program written in a low-level language. An interpreter is a program that performs a similar translation, but unlike a compiler, an interpreter executes a portion of code right after translating it, rather than translating the entire program at once.

- The Java compiler translates your Java program into a program in the bytecode language. When you give the command to run your Java program, this bytecode program is both translated into machine-language instructions and executed by an interpreter called the Java Virtual Machine.
- An object is a program construct that performs certain actions. These actions, or behaviors, are defined by the object's methods. The characteristics, or attributes, of an object are determined by its data, and the values of these attributes give the object a state.
- Object-oriented programming is a methodology that views a program as consisting of objects that can act alone or interact with one another. A software object might represent a real-world object, or it might be an abstraction.
- Three of the main principles of object-oriented programming are encapsulation, polymorphism, and inheritance.
- A class is a blueprint for the attributes and behaviors of a group of objects. The class defines the type of these objects. All objects of the same class have the same methods.
- In a Java program, a method invocation is written as the object name, followed by a period (called a dot), the method name, and, finally, the arguments in parentheses.
- An algorithm is a set of directions for solving a problem. To qualify as an algorithm, the directions must be expressed so completely and precisely that somebody could follow them without having to fill in any details or make any decisions that are not fully specified in the directions.
- Pseudocode is a combination of English and a programming language. It is used to write an algorithm's directions.
- The syntax of a programming language is the set of grammatical rules for the language. These rules dictate whether a statement in the language is correct. The compiler will detect errors in a program's syntax.
- You can write applets that display pictures on the computer screen. Applets are meant to be sent over the Internet and be viewed in a Web browser. However, you can use an applet viewer, which is a stand-alone program, instead.
- The method `drawOval` draws the outline of an oval. The method `fillOval` draws the same oval as `drawOval` but fills it in. The method `drawArc` draws an arc that is part of an oval.

Exercises

1. What is the use of the central processing unit in a computer? Name any one well-known processor.

2. After you use a text editor to write a program, will it be in main memory or auxiliary memory?

3. Name the supervisory program that oversees the entire operation of the computer?

4. How does machine language differ from assembly language?

5. How does bytecode differ from assembly language?

6. What would the following statements, when used in a Java program, display on the screen?

```
float salary;
salary = 5000.50f;
System.out.println ("My salary is");
System.out.println(salary);
```

7. Write a statement or statements that can be used in a Java program to display the following on the screen:

```
5
3
1
```

8. Write statements that can be used in a Java program to read your salary, as entered on the keyboard, and display it on the screen.

9. Given a person's year of joining in an organization and the current year, the Year of Service Wizard can compute the number of years the person has served. Write statements that can be used in a Java program to perform this computation for the Year of Service Wizard.

10. Write statements that can be used in a Java program to read two integers and display the number of even integers that lie between them. For example, the number of even integers that lie between 12 and 5 are 4.

11. A single bit can represent two values: 0 and 1. Two bits can represent four values: 00, 01, 10, and 11. Three bits can represent eight values: 000, 001, 010, 011, 100, 101, 110, and 111. How many values can be represented by

    a. 10 bits? **b.** 20 bits? **c.** 30 bits?

12. Find the documentation for the Java Class Library on the Oracle® Web site. (At this writing, the link to this documentation is http://download-llnw.oracle. com/javase/7/docs/api/.) Then find the description for the Package java.applet. How many Interfaces are described in the section entitled "Interface Summary"?

13. Self-Test Question 27 asked you to think of some attributes for a song object. What attributes would you want for an object that represents a cell phone number list containing many phone numbers?

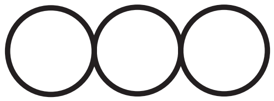
14. What behaviors might a cell phone number have? What behaviors might a cell phone number list have? Contrast the difference in behavior between the two kinds of objects.

15. What attributes and behaviors would an object representing a cell phone subscriber account have?

16. Suppose that you have a number x that is greater than 1. Write an algorithm that computes the largest integer k such that 3^k is less than or equal to x.

17. Write an algorithm that finds the minimum value in a list of values.

18. Write statements that can be used in a Java applet to draw the following three connected rings. (Don't worry about the color.)



Graphics

19. Find the documentation for the class Graphics2D in the Java Class Library. (See Exercise 12.) Learn how to use the method `draw3DRect`. Then explain the use of different parameters used in the method `draw3DRect`. Also explain the use of different parameters used in the method `fill3DRect`.

Graphics

20. Write statements that can be used in a Java applet to draw the outline of a crescent moon.

Graphics

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

1. Write the Java program shown in Listing 1.2 in the bin directory under the jdk directory. Name the file `HappyFace.java`. Compile the program so that you receive no compiler error messages. Then run the program.

2. Write a java program so that it adds four numbers. Compile the program so that you receive no compiler error messages. Then run the program.

3. The following program has syntax errors that prevent the program from compiling. Find and fix the errors.

```
import java.util.Scanner;
public class SyntaxError
{
  public static void main(String[] args)
  {
    System.out.println(“Enter two numbers to subtract”.);
    Scanner keyboard = new Scanner(System.in);
    n1 = keyboard.nextInt();
    n2 = keyboard.nextInt();
    int result = n1 - n2;
    System.out.println(The result is:  + result)
  }
}
```

4. The following program has syntax errors that prevent the program from compiling. Find and fix the errors.

```
import java.util.Scanner;
public class SemanticError
{
  public static void main(String[] args)
  {
    int width=0, depth=0;
    System.out.println(“Enter the height, width, and depth of”);
    System.out.println(“a box and I will compute the volume.”);
    Scanner keyboard = new Scanner(System.in);
    height = keyboard.nextInt();
    width = keyboard.nextInt();
    depth = keyboard.nextInt();
    int volume = height * width * depth;
    System.out.println(“The volume is “ + volume);
  }
}
```

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways.

1. Write a Java program that displays the following picture. (*Hint:* Write a sequence of `println` statements that display lines of asterisks and blanks.)

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

2. Write a complete program for the problem described in Exercise 9.

3. Write a complete program for the problem described in Exercise 10.

VideoNote Writing an algorithm for Project 3

4. Write an applet program similar to the one in Listing 1.2 that displays a picture of a snowman. (*Hint:* Draw three circles, one above the other. Make the circles progressively smaller from bottom to top. Make the top circle a happy face.)

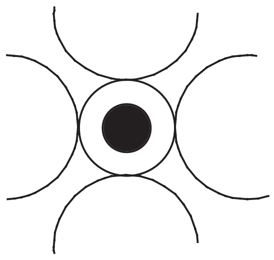
Graphics

5. Write an applet program for the problem described in Exercise 18.

Graphics

6. Write an applet program that displays the following pattern:

Graphics



Answers to Self-Test Questions

1. Main memory and auxiliary memory.

2. Software is just another name for programs.

3. The two numbers to be added.

4. All the grades on all the quizzes you have taken in the course.

5. A high-level-language program is written in a form that is easy for a human being to write and read. A machine-language program is written in a form

the computer can execute directly. A high-level-language program must be translated into a machine-language program before the computer can execute it. Java bytecode is a low-level language that is similar to the machine language of most common computers. It is relatively easy to translate a program expressed in Java bytecode into the machine language of almost any computer.

6. Java is a high-level language.

7. Java bytecode is a low-level language.

8. A compiler translates a high-level-language program into a low-level-language program such as a machine-language program or a Java bytecode program. When you compile a Java program, the compiler translates your Java program into a program expressed in Java bytecode.

9. A source program is the high-level-language program that is input to a compiler.

10. The Java Virtual Machine is a program that translates Java bytecode instructions into machine-language instructions. The JVM is a kind of interpreter.

11. `Java is great!`

12.
```
System.out.println("Java for one.");
System.out.println("Java for all.");
```

13. `mary.increaseAge(5);`

14. The statement reads a whole number typed in at the keyboard and stores it in the variable `n1`.

15.
```
public class Question15
{
   public static void main(String[] args)
   {
      System.out.println("Hello World!");
   }
}
```

Some details, such as identifier names, may be different in your program. Be sure you compile and run your program.

16. The file containing the class `YourClass` should be named `YourClass.java`.

17. `YourClass.class`.

18. A method defines an action that an object is capable of performing.

19. A class is a blueprint for creating objects. All objects in the same class have the same kind of data and the same methods.

20. Yes, all objects of the same class have the same methods.

21. Encapsulation is the process of hiding all the details of an object that are unnecessary to using the object. Put another way, encapsulation is the process of describing a class or object by giving only enough information to allow a programmer to use the class or object.

22. Information hiding is another term for encapsulation.

23. In a programming language, such as Java, polymorphism means that one method name, used as an instruction, can cause different actions, depending on the kind of object performing the action.

24. Inheritance is a way of organizing classes. You can define a general class having common attributes and behaviors, and then use inheritance to define specialized classes that add to or revise the details of the general class.

25. An algorithm is a set of directions for solving a problem. To qualify as an algorithm, the directions must be expressed so completely and precisely that somebody could follow them without having to fill in any details or make any decisions that are not fully specified in the directions.

26. Pseudocode is a mixture of English and Java that you can use to write the steps of an algorithm.

27. A song object could have the following attributes: title, composer, date, performer, album title.

28. Algorithm to count the odd integers in a list of integers:

    1. Write the number 0 on the blackboard.
    2. Do the following for each odd integer on the list:
        - Add 1 to the number on the blackboard.
        - Replace the old number on the blackboard with the result of this addition.
    3. Announce that the answer is the number written on the blackboard.

29. A syntax error is a grammatical mistake in a program. When you write a program, you must adhere to very strict grammatical rules. If you violate one of these rules by omitting a required punctuation mark, for example, you make a syntax error.

30. A logic error is a conceptual error in a program or its algorithm. If your program runs and gives output, but the output is incorrect, you have a logic error.

31. Syntax errors and run-time errors.

32. A logic error.

33. Change the following lines

```
canvas.fillOval(155, 100, 10, 20);
canvas.fillOval(230, 100, 10, 20);
```

to

```
canvas.fillOval(155, 100, 10, 10);
canvas.fillOval(230, 100, 10, 10);
```

The last two numbers on each line are changed from 10, 20 to 10, 10. You could also use some other number, such as 20, and write 20, 20 in place of 10, 10.

34. Change the following line

```
canvas.drawArc(150, 160, 100, 50, 180, 180);
```

to

```
canvas.drawArc(150, 160, 100, 50, 180, -180);
```

The last number is changed from positive to negative. Other correct answers are possible. For example, the following is also acceptable:

```
canvas.drawArc(150, 160, 100, 50, 0, 180);
```

You could also change the first number 150 to a larger number in either of the above statements. Other correct answers are similar to what we have already described.

Basic Computation 2

Beauty without expression tires.

—RALPH WALDO EMERSON, *The Conduct of Life*, (1876)

In this chapter, we explain enough about the Java language to allow you to write simple Java programs. You do not need any programming experience to understand this chapter. If you are already familiar with some other programming language, such as Visual Basic, C, C++, or C#, much that is in Section 2.1 will already be familiar to you. However, even if you know the concepts, you should learn the Java way of expressing them.

OBJECTIVES

After studying this chapter, you should be able to

- Describe the Java data types that are used for simple data like numbers and characters
- Write Java statements to declare variables and define named constants
- Write assignment statements and expressions containing variables and constants
- Define strings of characters and perform simple string processing
- Write Java statements that accomplish keyboard input and screen output
- Adhere to stylistic guidelines and conventions
- Write meaningful comments within a program
- Use the class `JFrame` to produce windowing interfaces within Java application programs
- Use the class `JOptionPane` to perform window-based input and output

PREREQUISITES

If you have not read Chapter 1, you should read at least the section of Chapter 1 entitled “A First Java Application Program” to familiarize yourself with the notions of class, object, and method. Also, material from the graphics supplement in Chapter 1 is used in the section “Style Rules Applied to a Graphics Applet” in the graphics supplement of this chapter.

2.1 VARIABLES AND EXPRESSIONS

In this section, we explain how simple variables and arithmetic expressions are used in Java programs. Some of the terms used here were introduced in Chapter 1. We will, however, review them again.

Variables

Variables in a program are used to store data such as numbers and letters. They can be thought of as containers of a sort. The number, letter, or other data item in a variable is called its **value.** This value can be changed, so that at one time the variable contains, say, 6, and at another time, after the program has run for a while, the variable contains a different value, such as 4.

A variable is a program component used to store or represent data

For example, the program in Listing 2.1 uses the variables `numberOfBaskets`, `eggsPerBasket`, and `totalEggs`. When this program is run, the statement

```
eggsPerBasket = 6;
```

sets the value of `eggsPerBasket` to 6.

Variables represent memory locations

In Java, variables are implemented as memory locations, which we described in Chapter 1. Each variable is assigned one memory location. When the variable is given a value, the value is encoded as a string of 0s and 1s and is placed in the variable's memory location.

LISTING 2.1 A Simple Java Program

```
public class EggBasket
{
    public static void main(String[] args)
    {
        int numberOfBaskets, eggsPerBasket, totalEggs;   <-- Variable declarations

        numberOfBaskets = 10;   <-- Assignment statement
        eggsPerBasket = 6;

        totalEggs = numberOfBaskets * eggsPerBasket;

        System.out.println("If you have");
        System.out.println(eggsPerBasket + " eggs per basket and");
        System.out.println(numberOfBaskets + " baskets, then");
        System.out.println("the total number of eggs is " + totalEggs);
    }
}
```

Sample Screen Output

```
If you have
6 eggs per basket and
10 baskets, then
the total number of eggs is 60
```

Choose meaningful variable names

You should choose variable names that are helpful. The names should suggest the variables' use or indicate the kind of data they will hold. For example, if you use a variable to count something, you might name it `count`. If the variable is used to hold the speed of an automobile, you might call the variable `speed`. You should almost never use single-letter variable names like `x` and `y`. Somebody reading the statement

```
x = y + z;
```

would have no idea of what the program is really adding. The names of variables must also follow certain spelling rules, which we will detail later in the section "Java Identifiers."

Before you can use a variable in your program, you must state some basic information about each one. The compiler—and so ultimately the computer—needs to know the name of the variable, how much computer memory to reserve for the variable, and how the data item in the variable is to be coded as strings of 0s and 1s. You give this information in a **variable declaration.** Every variable in a Java program must be declared before it is used for the first time.

A variable declaration tells the computer what type of data the variable will hold. That is, you declare the variable's data type. Since different types of data are stored in the computer's memory in different ways, the computer must know the type of a variable so it knows how to store and retrieve the value of the variable from the computer's memory. For example, the following line from Listing 2.1 declares `numberOfBaskets`, `eggsPerBasket`, and `totalEggs` to be variables of data type `int`:

```
int numberOfBaskets, eggsPerBasket, totalEggs;
```

A variable declaration consists of a type name, followed by a list of variable names separated by commas. The declaration ends with a semicolon. All the variables named in the list are declared to have the same data type, as given at the start of the declaration.

If the data type is `int`, the variable can hold whole numbers, such as 42, −99, 0, and 2001. A whole number is called an **integer.** The word `int` is an abbreviation of *integer*. If the type is `double`, the variable can hold numbers having a decimal point and a fractional part after the decimal point. If the type is `char`, the variables can hold any one character that appears on the computer keyboard.

Declare variables before using them

Every variable in a Java program must be declared before the variable can be used. Normally, a variable is declared either just before it is used or at the start of a section of your program that is enclosed in braces {}. In the simple programs we have seen so far, this means that variables are declared either just before they are used or right after the lines

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

RECAP Variable Declarations

In a Java program, you must declare a variable before it can be used. A variable declaration has the following form:

SYNTAX

```
Type Variable_1, Variable_2, ...;
```

EXAMPLES

```
int styleNumber, numberOfChecks, numberOfDeposits;
double amount, interestRate;
char answer;
```

Data Types

As you have learned, a data type specifies a set of values and their operations. In fact, the values have a particular data type because they are stored in memory in the same format and have the same operations defined for them.

A data type specifies a set of values and operations

REMEMBER Syntactic Variables

When you see something in this book like *Type*, *Variable_1,* or *Variable_2* used to describe Java syntax, these words do not literally appear in your Java code. They are **syntactic variables,** which are a kind of blank that you fill in with something from the category that they describe. For example, *Type* can be replaced by `int`, `double`, `char`, or any other type name. *Variable_1* and *Variable_2* can each be replaced by any variable name.

Java has two main kinds of data types: class types and primitive types. As the name implies, a **class type** is a data type for objects of a class. Since a class is like a blueprint for objects, the class specifies how the values of its type are stored and defines the possible operations on them. As we implied in the previous chapter, a class type has the same name as the class. For example, quoted strings such as `"Java is fun"` are values of the class type `String`, which is discussed later in this chapter.

Class types and primitive types

Variables of a **primitive type** are simpler than objects (values of a class type), which have both data and methods. A value of a primitive type is an indecomposable value, such as a single number or a single letter. The types `int`, `double`, and `char` are examples of primitive types.

FIGURE 2.1 Primitive Type

Type Name	Kind of Value	Memory Used	Range of Values
byte	Integer	1 byte	–128 to 127
short	Integer	2 bytes	–32,768 to 32,767
int	Integer	4 bytes	–2,147,483,648 to 2,147,483,647
long	Integer	8 bytes	–9,223,372,036,8547,75,808 to 9,223,372,036,854,775,807
float	Floating-point	4 bytes	$\pm 3.40282347 \times 10^{+38}$ to $\pm 1.40239846 \times 10^{-45}$
double	Floating-point	8 bytes	$\pm 1.79769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$
char	Single character (Unicode)	2 bytes	All Unicode values from 0 to 65,535
boolean		1 bit	True or false

Figure 2.1 lists all of Java's primitive types. Four types are for integers, namely, `byte`, `short`, `int`, and `long`. The only differences among the various integer types are the range of integers they represent and the amount of computer memory they use. If you cannot decide which integer type to use, use the type `int`.

A number having a fractional part—such as the numbers 9.99, 3.14159, −5.63, and 5.0—is called a **floating-point number.** Notice that 5.0 is a floating-point number, not an integer. If a number has a fractional part, even if the fractional part is zero, it is a floating-point number. As shown in Figure 2.1, Java has two data types for floating-point numbers, `float` and `double`. For example, the following code declares two variables, one of type `float` and one of type `double`:

A floating-point number has a fractional part

```
float cost;
double capacity;
```

As with integer types, the differences between `float` and `double` involve the range of their values and their storage requirements. If you cannot decide between the types `float` and `double`, use `double`.

The primitive type `char` is used for single characters, such as letters, digits, or punctuation. For example, the following declares the variable `symbol` to be of type `char`, stores the character for uppercase *A* in `symbol`, and then displays *A* on the screen:

```
char symbol;
symbol = 'A';
System.out.println(symbol);
```

In a Java program, we enclose a single character in single quotes, as in 'A'. Note that there is only one symbol for a single quote. The same quote symbol is used on both sides of the character. Finally, remember that uppercase letters and lowercase letters are different characters. For example, 'a' and 'A' are two different characters.

Single quotes enclose a character

The last primitive type we have to discuss is the type boolean. This data type has two values, true and false. We could, for example, use a variable of type boolean to store the answer to a true/false question such as "Is eggCount less than 12?" We will have more to say about the data type boolean in the next chapter.

All primitive type names in Java begin with a lowercase letter. In the next section, you will learn about a convention in which class type names—that is, the names of classes—begin with an uppercase letter.

Although you declare variables for class types and primitive types in the same way, these two kinds of variables store their values using different mechanisms. Chapter 5 will explain class type variables in more detail. In this chapter and the next two, we will concentrate on primitive types. We will occasionally use variables of a class type before Chapter 5, but only in contexts where they behave pretty much the same as variables of a primitive type.

Java Identifiers

The technical term for a name in a programming language, such as the name of a variable, is an **identifier.** In Java, an identifier (a name) can contain only letters, digits 0 through 9, and the underscore character (_). The first character in an identifier cannot be a digit.[1] In particular, no name can contain a space or any other character such as a dot (period) or an asterisk (*). There is no limit to the length of an identifier. Well, in practice, there is a limit, but Java has no official limit and will accept even absurdly long names. Java is **case sensitive.** That is, uppercase and lowercase letters are considered to be different characters. For example, Java considers mystuff, myStuff, and MyStuff to be three different identifiers, and you could have three different variables with these three names. Of course, writing variable names that differ only in their capitalization is a poor programming practice, but the Java compiler would happily accept them. Within these constraints, you can use any name you want for a variable, a class, or any other item you define in a Java program. But there are some style guidelines for choosing names.

Java is case sensitive

[1] Java does allow the dollar sign ($) to appear in an identifier, treating it as a letter. But such identifiers have a special meaning. It is intended to identify code generated by a machine, so you should not use the $ symbol in your identifiers.

Our somewhat peculiar use of uppercase and lowercase letters, such as `numberOfBaskets`, deserves some explanation. It would be perfectly legal to use `NumberOfBaskets` or `number_of_baskets` instead of `numberOfBaskets`, but these other names would violate some well-established conventions about how you should use uppercase and lowercase letters. Under these conventions, we write the names of variables using only letters and digits. We "punctuate" multiword names by using uppercase letters—since we cannot use spaces. The following are all legal names that follow these conventions:

Legal identifiers

```
inputStream YourClass CarWash hotCar theTimeOfDay
```

Notice that some of these legal names start with an uppercase letter and others, such as `hotCar`, start with a lowercase letter. We will always follow the convention that the names of classes start with an uppercase letter, and the names of variables and methods start with a lowercase letter.

The following identifiers are all illegal in Java, and the compiler will complain if you use any of them:

Illegal identifiers

```
prenhall.com go-team Five* 7eleven
```

The first three contain illegal characters, either a dot, a hyphen, or an asterisk. The last name is illegal because it starts with a digit.

Some words in a Java program, such as the primitive types and the word `if`, are called **keywords** or **reserved words.** They have a special predefined meaning in the Java language and cannot be used as the names of variables, classes, or methods, or for anything other than their intended meaning. All Java keywords are entirely in lowercase. A full list of these keywords appears in Appendix 11, which is online, and you will learn them as we go along. The program listings in this book show keywords, such as `public`, `class`, `static`, and `void`, in a special color. The text editors within an IDE often identify keywords in a similar manner.

Java keywords have special meanings

Some other words, such as `main` and `println`, have a predefined meaning but are not keywords. That means you can change their meaning, but it is a bad idea to do so, because it could easily confuse you or somebody else reading your program.

RECAP Identifiers (Names)

The name of something in a Java program, such as a variable, class, or method, is called an identifier. It must not start with a digit and may contain only letters, digits 0 through 9, and the underscore character (_). Uppercase and lowercase letters are considered to be different characters. (The symbol $ is also allowed, but it is reserved for special purposes, and so you should not use $ in a Java name.)

Although it is not required by the Java language, the common practice, and the one followed in this book, is to start the names of classes with uppercase letters and to start the names of variables and methods with lowercase letters. These names are usually spelled using only letters and digits.

GOTCHA Java Is Case Sensitive

Do not forget that Java is case sensitive. If you use an identifier, like `myNumber`, and then in another part of your program you use the spelling `MyNumber`, Java will not recognize them as being the same identifier. To be seen as the same identifier, they must use exactly the same capitalization. ■

FAQ Why should I follow naming conventions? And who sets the rules?

By following naming conventions, you can make your programs easier to read and to understand. Typically, your supervisor or instructor determines the conventions that you should follow when writing Java programs. However, the naming conventions that we just gave are almost universal among Java programmers. We will mention stylistic conventions for other aspects of a Java program as we go forward. Sun Microsystems provides its own conventions on its Web site. While the company suggests that all Java programmers follow these conventions, not everyone does.

Assignment Statements

The most straightforward way to give a variable a value or to change its value is to use an **assignment statement.** For example, if `answer` is a variable of type `int` and you want to give it the value 42, you could use the following assignment statement:

```
answer = 42;
```

The equal sign, `=`, is called the **assignment operator** when it is used in an assignment statement. It does not mean what the equal sign means in other contexts. The assignment statement is an order telling the computer to change the value stored in the variable on the left side of the assignment operator to the value of the **expression** on the right side. Thus, an assignment statement always consists of a single variable followed by the assignment operator (the equal sign) followed by an expression. The assignment statement ends with a semicolon. So assignment statements take the form

An assignment statement gives a value to a variable

```
Variable = Expression;
```

The expression can be another variable, a number, or a more complicated expression made up by using **arithmetic operators,** such as + and -, to combine variables and numbers. For example, the following are all examples of assignment statements:

```
amount = 3.99;
firstInitial = 'B';
score = numberOfCards + handicap;
eggsPerBasket = eggsPerBasket - 2;
```

All the names, such as amount, score, and numberOfCards, are variables. We are assuming that the variable amount is of type double, firstInitial is of type char, and the rest of the variables are of type int.

When an assignment statement is executed, the computer first evaluates the expression on the right side of the assignment operator (=) to get the value of the expression. It then uses that value to set the value of the variable on the left side of the assignment operator. You can think of the assignment operator as saying, "Make the value of the variable equal to what follows."

For example, if the variable numberOfCards has the value 7 and handicap has the value 2, the following assigns 9 as the value of the variable score:

```
score = numberOfCards + handicap;
```

In the program in Listing 2.1, the statement

* means multiply

```
totalEggs = numberOfBaskets * eggsPerBasket;
```

is another example of an assignment statement. It tells the computer to set the value of totalEggs equal to the number in the variable numberOfBaskets multiplied by the number in the variable eggsPerBasket. The asterisk character (*) is the symbol used for multiplication in Java.

Note that a variable can meaningfully occur on both sides of the assignment operator and can do so in ways that might at first seem a little strange. For example, consider

The same variable can occur on both sides of the =

```
count = count + 10;
```

This does not mean that the value of count is equal to the value of count plus 10, which, of course, is impossible. Rather, the statement tells the computer to add 10 to the *old* value of count and then make that the *new* value of count. In effect, the statement will increase the value of count by 10. Remember that when an assignment statement is executed, the computer first evaluates the expression on the right side of the assignment operator and then makes that result the new value of the variable on the left side of the assignment operator. As another example, the following assignment statement will decrease the value of eggsPerBasket by 2:

```
eggsPerBasket = eggsPerBasket - 2;
```

RECAP Assignment Statements Involving Primitive Types

An assignment statement that has a variable of a primitive type on the left side of the equal sign causes the following action: First, the expression on the right side of the equal sign is evaluated, and then the variable on the left side of the equal sign is set to this value.

SYNTAX

```
Variable = Expression;
```

EXAMPLE

```
score = goals - errors;
interest = rate * balance;
number = number + 5;
```

PROGRAMMING TIP Initialize Variables

A variable that has been declared, but that has not yet been given a value by an assignment statement (or in some other way), is said to be **uninitialized.** If the variable is a variable of a class type, it literally has no value. If the variable has a primitive type, it likely has some default value. However, your program will be clearer if you explicitly give the variable a value, even if you are simply reassigning the default value. (The exact details on default values have been known to change and should not be counted on.)

You can initialize a variable when you declare it

One easy way to ensure that you do not have an uninitialized variable is to initialize it within the declaration. Simply combine the declaration and an assignment statement, as in the following examples:

```
int count = 0;
double taxRate = 0.075;
char grade = 'A';
int balance = 1000, newBalance;
```

Note that you can initialize some variables and not initialize others in a declaration.

Sometimes the compiler may complain that you have failed to initialize a variable. In most cases, that will indeed be true. Occasionally, though, the compiler is mistaken in giving this advice. However, the compiler will not compile your program until you convince it that the variable in question is initialized. To make the compiler happy, initialize the variable when you declare it, even if the variable will be given another value before it is used for anything. In such cases, you cannot argue with the compiler. ■

RECAP Combining a Variable Declaration and an Assignment

You can combine the declaration of a variable with an assignment statement that gives the variable a value.

SYNTAX

```
Type Variable_1 = Expression_1, Variable_2 = Expression_2,
. . .;
```

EXAMPLES

```
int numberSeen = 0, increment = 5;
double height = 12.34, prize = 7.3 + increment;
char answer = 'y';
```

Simple Input

In Listing 2.1, we set the values of the variables `eggsPerBasket` and `numberOfBaskets` to specific numbers. It would make more sense to obtain the values needed for the computation from the user, so that the program could be run again with different numbers. Listing 2.2 shows a revision of the program in Listing 2.1 that asks the user to enter numbers as input at the keyboard.

Use the standard class Scanner to accept keyboard input

We use the class `Scanner`, which Java supplies, to accept keyboard input. Our program must import the definition of the `Scanner` class from the package `java.util`. Thus, we begin the program with the following statement:

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

The following line sets things up so that data can be entered from the keyboard:

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

This line must appear before the first statement that takes input from the keyboard. That statement in our example is

```
eggsPerBasket = keyboard.nextInt();
```

This assignment statement gives a value to the variable `eggsPerBasket`. The expression on the right side of the equal sign, namely

```
keyboard.nextInt()
```

reads one `int` value from the keyboard. The assignment statement makes this `int` value the value of the variable `eggsPerBasket`, replacing any value that the variable might have had. When entering numbers at the keyboard, the user must either separate multiple numbers with one or more spaces or place each number on its own line. Section 2.3 will explain such keyboard input in detail.

LISTING 2.2 A Program with Keyboard Input

```
import java.util.Scanner;

public class EggBasket2
{
   public static void main(String[] args)
   {
      int numberOfBaskets, eggsPerBasket, totalEggs;

      Scanner keyboard = new Scanner(System.in);
      System.out.println("Enter the number of eggs in each basket:");
      eggsPerBasket = keyboard.nextInt();
      System.out.println("Enter the number of baskets:");
      numberOfBaskets = keyboard.nextInt();

      totalEggs = numberOfBaskets * eggsPerBasket;

      System.out.println("If you have");
      System.out.println(eggsPerBasket + " eggs per basket and");
      System.out.println(numberOfBaskets + " baskets, then");
      System.out.println("the total number of eggs is " + totalEggs);

      System.out.println("Now we take two eggs out of each basket.");

      eggsPerBasket = eggsPerBasket - 2;
      totalEggs = numberOfBaskets * eggsPerBasket;

      System.out.println("You now have");
      System.out.println(eggsPerBasket + " eggs per basket and");
      System.out.println(numberOfBaskets + " baskets.");
      System.out.println("The new total number of eggs is " + totalEggs);
   }
}
```

Gets the `Scanner` *class from the package (library)* `java.util`

Sets up things so the program can accept keyboard input

Reads one whole number from the keyboard

Sample Screen Output

```
Enter the number of eggs in each basket:
6
Enter the number of baskets:
10
If you have
6 eggs per basket and
10 baskets, then
the total number of eggs is 60
Now we take two eggs out of each basket.
You now have
4 eggs per basket and
10 baskets.
The new total number of eggs is 40
```

Simple Screen Output

Now we will give you a brief overview of screen output—just enough to allow you to write and understand programs like the one in Listing 2.2. `System` is a class that is part of the Java language, and `out` is a special object within that class. The object `out` has `println` as one of its methods. It may seem strange to write `System.out.println` to call a method, but that need not concern you at this point. Chapter 6 will provide some details about this notation.

So

```
System.out.println(eggsPerBasket + "eggs per basket.");
```

displays the value of the variable `eggsPerBasket` followed by the phrase *eggs per basket.* Notice that the + symbol does not indicate arithmetic here. It denotes another kind of "and." You can read the preceding Java statement as an instruction to display the value of the variable `eggsPerBasket` and then to display the string `"eggs per basket."`

Section 2.3 will continue the discussion of screen output.

Constants

A variable can have its value changed. That is why it is called a variable: Its value *varies*. A number like 2 cannot change. It is always 2. It is never 3. In Java, terms like `2` or `3.7` are called **constants,** or **literals,** because their values do not change.

A constant does not change in value

Constants need not be numbers. For example, `'A'`, `'B'`, and `'$'` are three constants of type `char`. Their values cannot change, but they can be used in an assignment statement to change the value of a variable of type `char`. For example, the statement

```
firstInitial = 'B';
```

changes the value of the char variable `firstInitial` to `'B'`.

There is essentially only one way to write a constant of type `char`, namely, by placing the character between single quotes. On the other hand, some of the rules for writing numeric constants are more involved. Constants of integer types are written the way you would expect them to be written, such as `2`, `3`, `0`, `−3`, or `752`. An integer constant can be prefaced with a plus sign or a minus sign, as in `+12` and `−72`. Numeric constants cannot contain commas. The number 1,000 is *not* correct in Java. Integer constants cannot contain a decimal point. A number with a decimal point is a floating-point number.

Floating-point constant numbers may be written in either of two forms. The simple form is like the everyday way of writing numbers with digits after the decimal point. For example, `2.5` is a floating-point constant. The other, slightly more complicated form is similar to a notation commonly used in mathematics and the physical sciences, **scientific notation.** For instance, consider the number 865000000.0. This number can be expressed more clearly in the following scientific notation:

Java's e notation is like scientific notation

```
8.65 × 10^8
```

Java has a similar notation, frequently called either **e notation** or **floating-point notation.** Because keyboards have no way of writing exponents, the 10 is omitted and both the multiplication sign and the 10 are replaced by the letter e. So in Java, 8.65×10^8 is written as `8.65e8`. The e stands for *exponent,* since it is followed by a number that is thought of as an exponent of 10. This form and the less convenient form `865000000.0` are equivalent in a Java program. Similarly, the number 4.83×10^{-4}, which is equal to 0.000483, could be written in Java as either `0.000483` or `4.83e-4`. Note that you also could write this number as `0.483e-3` or `48.3e-5`. Java does not restrict the position of the decimal point.

VideoNote
Another sample program

The number before the e may contain a decimal point, although it doesn't have to. The number after the e cannot contain a decimal point. Because multiplying by 10 is the same as moving the decimal point in a number, you can think of a positive number after the e as telling you to move the decimal point that many digits to the right. If the number after the e is negative, you move the decimal point that many digits to the left. For example, `2.48e4` is the same number as `24800.0`, and `2.48e-2` is the same number as `0.0248`.

FAQ What is "floating" in a floating-point number?

Floating-point numbers got their name because, with the e notation we just described, the decimal point can be made to "float" to a new location by adjusting the exponent. You can make the decimal point in `0.000483` float to after the 4 by expressing this number as the equivalent expression `4.83e-4`. Computer language implementers use this trick to store each floating-point number as a number with exactly one digit before the decimal point (and some suitable exponent). Because the implementation always floats the decimal point in these numbers, they are called floating-point numbers. Actually, the numbers are stored in another base, such as 2 or 16, rather than as the decimal (base 10) numbers we used in our example, but the principle is the same.

FAQ Is there an actual difference between the constants `5` and `5.0`?

The numbers 5 and 5.0 are conceptually the same number. But Java considers them to be different. Thus, `5` is an integer constant of type `int`, but `5.0` is a floating-point constant of type `double`. The number 5.0 contains a fractional part, even though the fraction is 0. Although you might see the numbers 5 and 5.0 as having the same value, Java stores them differently. Both integers and floating-point numbers contain a finite number of digits when stored in a computer, but only integers are considered exact quantities. Because floating-point numbers have a fractional portion, they are seen as approximations.

GOTCHA Imprecision in Floating-Point Numbers

Floating-point numbers are stored with a limited amount of precision and so are, for all practical purposes, only approximate quantities. For example, the fraction one third is equal to

```
0.3333333 . . .
```

where the three dots indicate that the 3s go on forever. The computer stores numbers in a format somewhat like the decimal representation on the previously displayed line, but it has room for only a limited number of digits. If it can store only ten digits after the decimal, then one third is stored as

```
0.3333333333 (and no more 3s)
```

This number is slightly smaller than one third and so is only approximately equal to one third. In reality, the computer stores numbers in binary notation, rather than in base 10, but the principles are the same and the same sorts of things happen.

Not all floating-point numbers lose accuracy when they are stored in the computer. Integral values like 29.0 can be stored exactly in floating-point notation, and so can some fractions like one half. Even so, we usually will not know whether a floating-point number is exact or an approximation. When in doubt, assume that floating-point numbers are stored as approximate quantities. ■

Named Constants

Java provides a mechanism that allows you to define a variable, initialize it, and moreover fix the variable's value so that it cannot be changed. The syntax is

Name important constants

```
public static final Type Variable = Constant;
```

For example, we can give the name `PI` to the constant `3.14159` as follows:

```
public static final double PI = 3.14159;
```

You can simply take this as a long, peculiarly worded way of giving a name (like `PI`) to a constant (like `3.14159`), but we can explain most of what is on this line. The part

```
double PI = 3.14159;
```

simply declares `PI` as a variable and initializes it to 3.14159. The words that precede this modify the variable `PI` in various ways. The word `public` says that there are no restrictions on where you can use the name `PI`. The word `static` will have to wait until Chapter 6 for an explanation; for now, just be sure to include it. The word `final` means that the value 3.14159 is the final value assigned to `PI` or, to phrase it another way, that the program is not allowed to change the value of `PI`.

The convention for naming constants is to use all uppercase letters, with an underscore symbol (_) between words. For example, in a calendar program, you might define the following constant:

```
public static final int DAYS_PER_WEEK = 7;
```

Although this convention is not required by the definition of the Java language, most programmers adhere to it. Your programs will be easier to read if you can readily identify variables, constants, and so forth.

RECAP Named Constants

To define a name for a constant, write the keywords `public static final` in front of a variable declaration that includes the constant as the initializing value. Place this declaration within the class definition but outside of any method definitions, including the `main` method.

SYNTAX

```
public static finalType Variable = Constant;
```

EXAMPLES

```
public static final int MAX_STRIKES = 3;
public static final double MORTGAGE_INTEREST_RATE = 6.99;
public static final String MOTTO =
                    "The customer is right!";
public static final char SCALE = 'K';
```

Although it is not required, most programmers spell named constants using all uppercase letters, with an underscore to separate words.

Assignment Compatibilities

As the saying goes, "You can't put a square peg in a round hole," and you can't put a `double` value like 3.5 in a variable of type `int`. You cannot even put the `double` value 3.0 in a variable of type `int`. You cannot store a value of one type in a variable of another type unless the value is somehow converted to match the type of the variable. However, when dealing with numbers, this conversion will sometimes—but not always—be performed automatically for you. The conversion will always be done when you assign a value of an integer type to a variable of a floating-point type, such as

```
double doubleVariable = 7;
```

Slightly more subtle assignments, such as the following, also perform the conversion automatically:

```
int intVariable = 7;
double doubleVariable = intVariable;
```

A value can be assigned to a variable whose type allows more precision

More generally, you can assign a value of any type in the following list to a variable of any type that appears further down in the list:

byte → short → int → long → float → double

For example, you can assign a value of type `long` to a variable of type `float` or to a variable of type `double` (or, of course, to a variable of type `long`), but you cannot assign a value of type `long` to a variable of type `byte`, `short`, or `int`. Note that this is not an arbitrary ordering of the types. As you move down the list from left to right, the types become more precise, either because they allow larger values or because they allow decimal points in the numbers. Thus, you can store a value into a variable whose type allows more precision than the type of the value allows.

In addition, you can assign a value of type `char` to a variable of type `int` or to any of the numeric types that follow `int` in our list of types. This particular assignment compatibility will be important when we discuss keyboard input. However, we do not advise assigning a value of type `char` to a variable of type `int` except in certain special cases.[2]

If you want to assign a value of type `double` to a variable of type `int`, you must change the type of the value using a type cast, as we explain in the next section.

RECAP Assignment Compatibilities

You can assign a value of any type on the following list to a variable of any type that appears further down on the list:

byte → short → int → long → float → double

In particular, note that you can assign a value of any integer type to a variable of any floating-point type.

It is also legal to assign a value of type `char` to a variable of type `int` or to any of the numeric types that follow `int` in our list of types.

[2] Readers who have used certain other languages, such as C or C++, may be surprised to learn that you cannot assign a value of type `char` to a variable of type `byte`. This is because Java reserves two bytes of memory for each value of type `char` but naturally reserves only one `byte` of memory for values of type `byte`.

Type Casting

The title of this section has nothing to do with the Hollywood notion of typecasting. In fact, it is almost the opposite. In Java—and in most programming languages—a **type cast** changes the data type of a value from its normal type to some other type. For example, changing the type of the value 2.0 from `double` to `int` involves a type cast. The previous section described when you can assign a value of one type to a variable of another type and have the type conversion occur automatically. In all other cases, if you want to assign a value of one type to a variable of another type, you must perform a type cast. Let's see how this is done in Java.

Suppose you have the following:

```
double distance = 9.0;
int points = distance;   <-- This assignment is illegal.
```

As the note indicates, the last statement is illegal in Java. You cannot assign a value of type `double` to a variable of type `int`, even if the value of type `double` happens to have all zeros after the decimal point and so is conceptually a whole number.

In order to assign a value of type `double` to a value of type `int`, you must place `(int)` in front of the value or the variable holding the value. For example, you can replace the preceding illegal assignment with the following and get a legal assignment:

A type cast changes the data type of a value

```
int points = (int)distance;   <-- This assignment is legal.
```

The expression `(int)distance` is called a type cast. Neither `distance` nor the value stored in `distance` is changed in any way. But the value stored in `points` is the "`int` version" of the value stored in `distance`. If the value of `distance` is 25.36, the value of `(int)distance` is 25. So `points` contains 25, but the value of `distance` is still 25.36. If the value of `distance` is 9.0, the value assigned to `points` is 9, and the value of `distance` remains unchanged.

An expression like `(int) 25.36` or `(int)distance` is an expression that *produces* an `int` value. A type cast does not change the value of the source variable. The situation is analogous to computing the number of (whole) dollars you have in an amount of money. If you have $25.36, the number of dollars you have is 25. The $25.36 has not changed; it has merely been used to produce the whole number 25.

For example, consider the following code:

```
double dinnerBill = 25.36;
int dinnerBillPlusTip = (int)dinnerBill + 5;
System.out.println("The value of dinnerBillPlusTip is " +
   dinnerBillPlusTip);
```

Truncation discards the fractional part

The expression (int)dinnerBill produces the value 25, so the output of this code would be

```
The value of dinnerBillPlusTip is 30
```

But the variable dinnerBill still contains the value 25.36.

Be sure to note that when you type cast from a double to an int—or from any floating-point type to any integer type—the amount is not rounded. The part after the decimal point is simply discarded. This is known as **truncating.** For example, the following statements

```
double dinnerBill = 26.99;
int numberOfDollars = (int)dinnerBill;
```

set numberOfDollars to 26, not 27. The result is *not rounded*.

As we mentioned previously, when you assign an integer value to a variable of a floating-point type—double, for example—the integer is automatically type cast to the type of the variable. For example, the assignment statement

```
double point = 7;
```

is equivalent to

```
double point = (double)7;
```

The type cast (double) is implicit in the first version of the assignment. The second version, however, is legal.

RECAP Type Casting

In many situations, you cannot store a value of one type in a variable of another type unless you use a type cast that converts the value to an equivalent value of the target type.

SYNTAX

```
(Type_Name)Expression
```

EXAMPLES

```
double guess = 7.8;
int answer = (int)guess;
```

The value stored in answer will be 7. Note that the value is truncated, not rounded. Note also that the variable *guess* is not changed in any way; it still contains 7.8. The last assignment statement affects only the value stored in *answer*.

PROGRAMMING TIP Type Casting a Character to an Integer

Java sometimes treats values of type `char` as integers, but the assignment of integers to characters has no connection to the meaning of the characters. For example, the following type cast will produce the `int` value corresponding to the character `'7'`:

```
char symbol = '7';
System.out.println((int)symbol);
```

You might expect the preceding to display 7, but it does not. It displays the number 55. Java, like all other programming languages, uses an arbitrary numbering of characters to encode them. Thus, each character corresponds to an integer. In this correspondence, the digits 0 through 9 are characters just like the letters or the plus sign. No effort was made to have the digits correspond to their intuitive values. Basically, they just wrote down all the characters and then numbered them in the order they were written down. The character `'7'` just happened to get 55. This numbering system is called the Unicode system, which we discuss later in the chapter. If you have heard of the ASCII numbering system, the Unicode system is the same as the ASCII system for the characters in the English language.

SELF-TEST QUESTIONS

1. Which of the following may be used as variable names in Java?

   ```
   rate1, 1stPlayer, myprogram.java, long, TimeLimit,
   numberOfWindows
   ```

2. Can a Java program have two different variables with the names `aVariable` and `avariable`?

3. Give the declaration for a variable called `count` of type `int`. The variable should be initialized to zero in the declaration.

4. Give the declaration for two variables of type `double`. The variables are to be named `rate` and `time`. Both variables should be initialized to zero in the declaration.

5. Write the declaration for two variables called `miles` and `flowRate`. Declare the variable `miles` to be of type `int` and initialize it to zero in the declaration. Declare the variable `flowRate` to be of type `double` and initialize it to 50.56 in the declaration.

6. What is the normal spelling convention for named constants?

7. Give a definition for a named constant for the number of hours in a day.

8. Write a Java assignment statement that will set the value of the variable `interest` to the value of the variable `balance` multiplied by 0.05.

9. Write a Java assignment statement that will set the value of the variable `interest` to the value of the variable `balance` multiplied by the value of the variable `rate`. The variables are of type `double`.

10. Write a Java assignment statement that will increase the value of the variable `count` by 3. The variable is of type `int`.

11. What is the output produced by the following lines of program code?

```
char a, b;
a = 'b';
System.out.println(a);
b = 'c';
System.out.println(b);
a = b;
System.out.println(a);
```

12. In the Programming Tip entitled "Type Casting a Character to an Integer," you saw that the following does not display the integer 7:

```
char symbol = '7';
System.out.println((int)symbol);
```

Thus, `(int)symbol` does not produce the number corresponding to the digit in `symbol`. Can you write an expression that will work to produce the integer that intuitively corresponds to the digit in `symbol`, assuming that `symbol` contains one of the ten digits 0 through 9? (*Hint*: The digits do correspond to consecutive integers, so if `(int)'7'` is 55, then `(int)'8'` is 56.)

Arithmetic Operators

In Java, you can perform arithmetic involving addition, subtraction, multiplication, and division by using the arithmetic operators +, −, *, and /, respectively. You indicate arithmetic in basically the same way that you do in ordinary arithmetic or algebra. You can combine variables or numbers—known collectively as **operands**—with these operators and parentheses to form an **arithmetic expression.** Java has a fifth arithmetic operator, %, that we will define shortly.

An arithmetic expression combines operands, operators, and parentheses

The meaning of an arithmetic expression is basically what you expect it to be, but there are some subtleties about the type of the result and, occasionally, even about the value of the result. All five of the arithmetic operators can be used with operands of any of the integer types, any of the floating-point types, and even with operands of differing types. The type of the value produced depends on the types of the operands being combined.