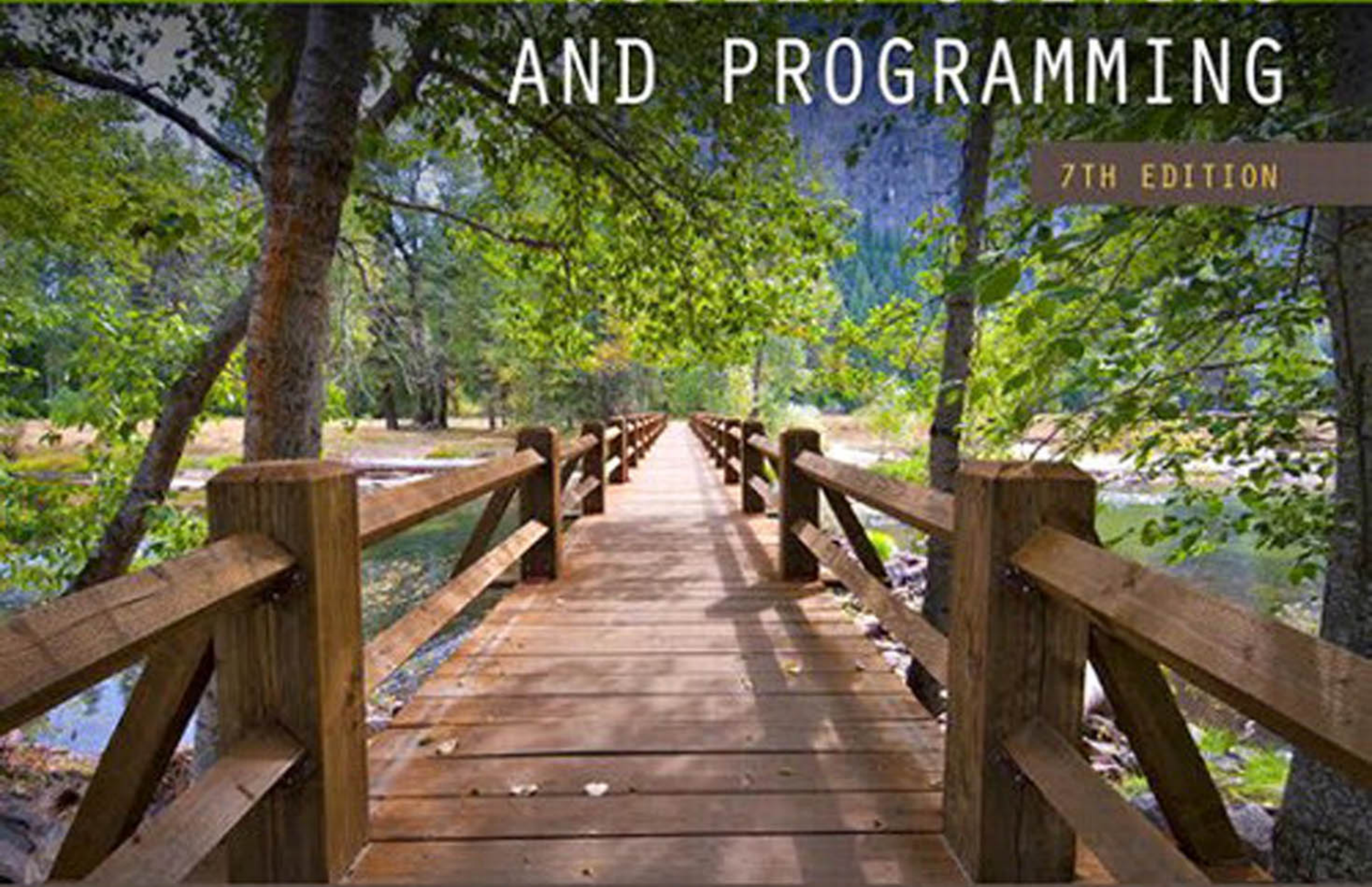
JAVA™

AN INTRODUCTION TO

PROBLEM SOLVING AND PROGRAMMING

7TH EDITION



WALTER SAVITCH

Java™

An Introduction to Problem Solving & Programming

This page intentionally left blank

Java™

7th edition

An Introduction to
Problem Solving & Programming

Walter Savitch
University of California, San Diego

Contributor
Kenrick Mock
University of Alaska Anchorage

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*
Senior Operations Supervisor: *Vincent Scelta*
Operations Specialist: *Linda Sager*
Art Director: *Kristine Carney*
Cover Designer: *Joyce Wells*
Manager, Rights and Permissions: *Michael Joyce*
Media Director: *Daniel Sandin*
Full-Service Project Management: *Hardik Popli, Cenveo® Publisher Services*
Composition: *Cenveo Publisher Services*
Interior Printer/Bindery: *Courier/Westford*
Cover Printer: *Lehigh-Phoenix Color/Hagerstown*

Library of Congress Cataloging-in-Publication Data

Savitch, Walter J., 1943-
Java : an introduction to problem solving and programming / Walter Savitch. -- 7th edition.
pages cm
Includes index.
ISBN-13: 978-0-13-376626-4 (alkaline paper)
ISBN-10: 0-13-376626-8 (alkaline paper)
1. Java (Computer program language) I. Title.
QA76.73.J38S27 2014
005.13’3--dc23
2013050398

10 9 8 7 6 5 4 3 2 1—CW—15 14 13 12 11

PEARSON

ISBN 10: 0-13-376626-8
ISBN 13: 978-0-13-376626-4

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™ 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.pearsonhighered.com/irc:

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 pearsonhighered.com/replocator.

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.pearsonhighered.com/savitch to access the student resources.

Online Practice and Assessment with MyProgrammingLab

MyProgrammingLab helps students fully grasp the logic, semantics, and syntax of programming. Through practice exercises and immediate, personalized feedback, MyProgrammingLab improves the programming competence of beginning students who often struggle with the basic concepts and paradigms of popular high-level programming languages.

A self-study and homework tool, a MyProgrammingLab course consists of hundreds of small practice problems organized around the structure of this textbook. For students, the system automatically detects errors in the logic and syntax of their code submissions and offers targeted hints that enable students to figure out what went wrong—and why. For instructors, a comprehensive gradebook tracks correct and incorrect answers and stores the code inputted by students for review.

MyProgrammingLab is offered to users of this book in partnership with Turing's Craft, the makers of the CodeLab interactive programming exercise system. For a full demonstration, to see feedback from instructors and students, or to get started using MyProgrammingLab in your course, visit www.myprogramminglab.com.

VideoNotes

VideoNote

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.pearsonhighered.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.

Online Practice with MyProgrammingLab

A self-study and practice tool, a MyProgrammingLab course consists of hundreds of small practice problems organized around the structure of this textbook. The system automatically detects errors in the logic and syntax of your code submissions and offers targeted hints that enable you to figure out what went wrong—and why. Visit www.myprogramminglab.com for more information.

VideoNotes

VideoNote

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.

This page intentionally left blank

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*

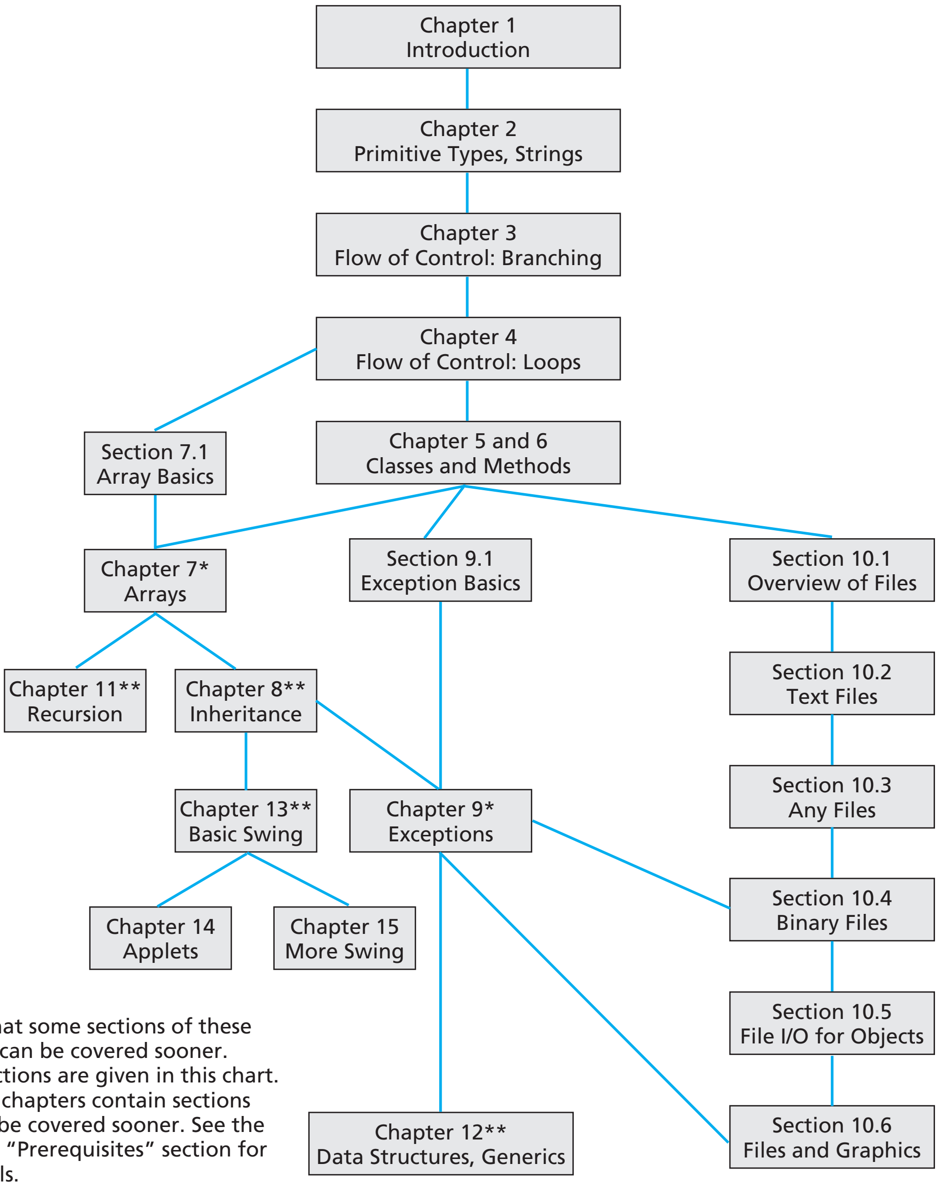
Jim Buffenbarger—*Boise State University*
Joan Boone—*University of North Carolina at Chapel Hill*
John Motil—*California State University, Northridge*
Ken Slonneger—*University of Iowa*
Laird Dornan—*Sun Microsystems, Inc.*
Le Gruenwald—*University of Oklahoma*
Lily Hou—*Carnegie Mellon University*
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.

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.

Remembers

Highlight important ideas that students should keep in mind.

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.

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[1] [1] 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.

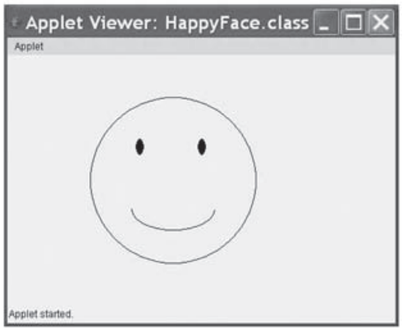
Listings

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*

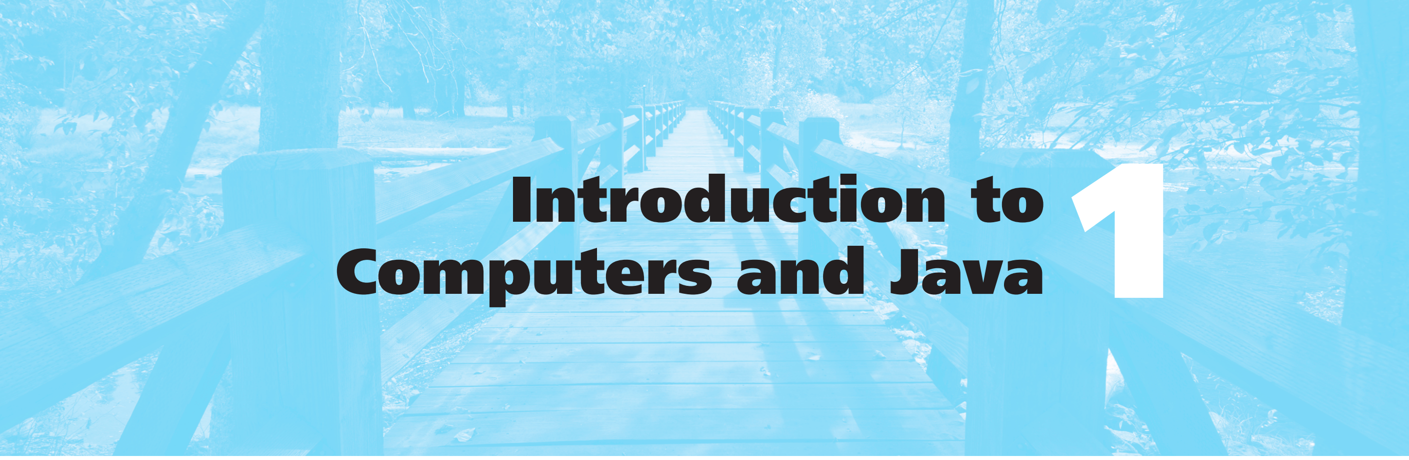
Our use of the terms *parameter* and *argument* is consistent with common usage. We use *parameter* to describe the definition of the data type and variable inside the header of a method and *argument* to describe items passed into a method when it is invoked. However, people often use these terms interchangeably. 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 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.

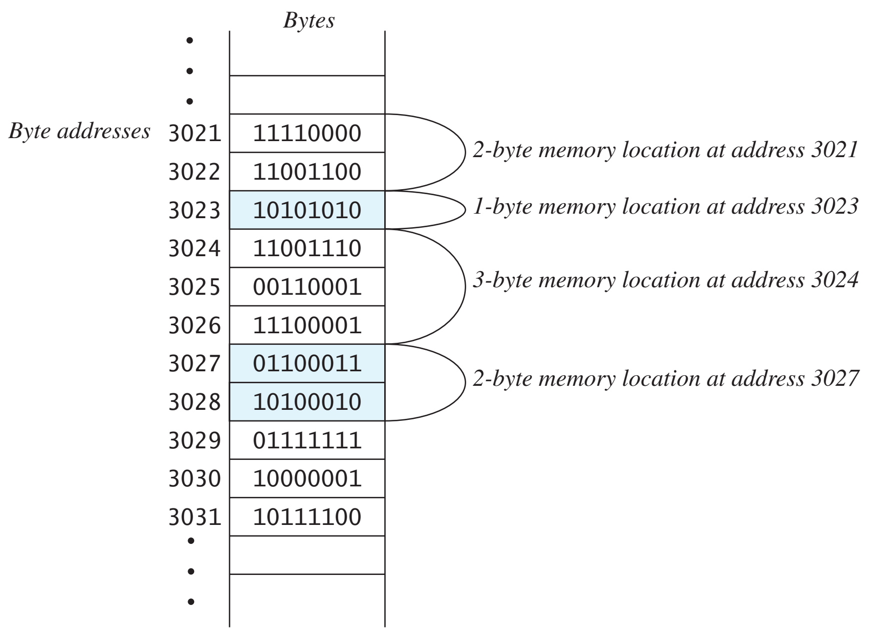
Main memory consists of addressable eight-bit bytes

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.

Groups of adjacent bytes can serve as a single memory location

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.

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 `0`s and `1`s. 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

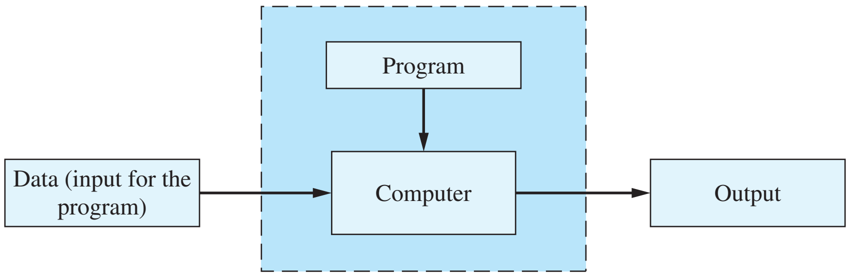
You probably have some idea of what a program is. You use programs all the time. For example, text editors and word processors are programs. As we mentioned earlier, a program is simply a set of instructions for a computer to follow. When you give the computer a program and some data and tell the computer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

A compiler translates Java code into 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.

The JVM is an interpreter that translates and executes 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.

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.

Java bytecode runs on any computer that has a JVM

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.

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.