Includes
Java 8
coverage

2/e

Cay Horstmann

Big Java

Late Objects

WILEY

Variable and Constant Declarations

```
  Type      Name       Initial value
  /         /          /
int cansPerPack = 6;

final double CAN_VOLUME = 0.335;
```

Method Declaration

```
                                           Parameter
   Modifiers          Return type          type and name
    /    \            /                    /        /
public static double cubeVolume(double sideLength)
{
   double volume = sideLength * sideLength * sideLength;
   return volume;
}                         Exits method and
                           returns result.
```

Mathematical Operations

`Math.pow(x, y)`	Raising to a power x^y
`Math.sqrt(x)`	Square root $\sqrt{x}$
`Math.log10(x)`	Decimal log $\log_{10}(x)$
`Math.abs(x)`	Absolute value $\|x\|$

`Math.sin(x)`
`Math.cos(x)`
`Math.tan(x)`
	Sine, cosine, tangent of x (x in radians)

Selected Operators and Their Precedence

(See Appendix B for the complete list.)

`[]`	Array element access
`++ -- !`	Increment, decrement, Boolean *not*
`* / %`	Multiplication, division, remainder
`+ -`	Addition, subtraction
`< <= > >=`	Comparisons
`== !=`	Equal, not equal
`&&`	Boolean *and*
`||`	Boolean *or*
`=`	Assignment

String Operations

```
String s = "Hello";
int n = s.length(); // 5
char ch = s.charAt(1); // 'e'
String t = s.substring(1, 4); // "ell"
String u = s.toUpperCase(); // "HELLO"
if (u.equals("HELLO")) ... // Use equals, not ==
for (int i = 0; i < s.length(); i++)
{
   char ch = s.charAt(i);
   Process ch
}
```

Conditional Statement

```
if (floor >= 13)                  Condition
{
   actualFloor = floor - 1;      ] Executed when condition is true
}
else if (floor >= 0)              Second condition (optional)
{
   actualFloor = floor;
}
else
{                                            Executed when
   System.out.println("Floor negative");  ]  all conditions are
}                                            false (optional)
```

Class Declaration

```
public class BankAccount
{
   private double balance;        ] Instance variables
   private int transactions;

   public BankAccount(double initialBalance)  ]
   {                                           ]  Constructor
      balance = initialBalance;                ]
      transactions = 1;                        ]
   }

   public void deposit(double amount)  ]
   {                                    ]  Method
      balance = balance + amount;       ]
      transactions++;                   ]
   }
   . . .
}
```

Loop Statements

```
                  Condition
                  /
while (balance < TARGET)
{
   year++;                                ]  Executed while
   balance = balance * (1 + rate / 100);  ]  condition is true
}
```

```
                  Loop body executed
do                  at least once
{
   System.out.print("Enter a positive integer: ");
   input = in.nextInt();
}
while (input <= 0);
```

```
     Initialization  Condition  Update
          /             /        /
for (int i = 0; i < 10; i++)
{
   System.out.println(i);
}
```

```
Set to a new element in each iteration
                                An array or collection
                                /
for (double value : values)
{
   sum = sum + value;   ]  Executed for each element
}
```

2/e

Big Java

Late Objects

Cay Horstmann

San Jose State University

WILEY

PUBLISHER	Laurie Rosatone
EDITORIAL DIRECTOR	Don Fowley
DEVELOPMENTAL EDITOR	Cindy Johnson
ASSISTANT DEVELOPMENT EDITOR	Ryann Dannelly
EXECUTIVE MARKETING MANAGER	Dan Sayre
SENIOR PRODUCTION EDITOR	Laura Abrams
SENIOR CONTENT MANAGER	Valerie Zaborski
EDITORIAL ASSISTANT	Anna Pham
SENIOR DESIGNER	Tom Nery
SENIOR PHOTO EDITOR	Billy Ray
PRODUCTION MANAGEMENT	Cindy Johnson
COVER IMAGES	(elephant) bluefem © Getty Images/iStock Collection; (tiger) © Aprison Photography/Getty Images, Inc.; (bird) © Nen-gloveyou/Shutterstock; (monkey) © Ehlers/iStockphoto.

This book was set in Stempel Garamond LT Std by Publishing Services, and printed and bound by Quad/Graphics, Versailles. The cover was printed by Quad/Graphics, Versailles.

This book is printed on acid-free paper. ∞

ISBN 13: 978-1-119-32107-1

The inside back cover will contain printing identification and country of origin if omitted from this page. In addition, if the ISBN on the back cover differs from the ISBN on this page, the one on the back cover is correct.

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

PREFACE

This book is an introduction to Java and computer programming that focuses on the essentials—and on effective learning. The book is designed to serve a wide range of student interests and abilities and is suitable for a first course in programming for computer scientists, engineers, and students in other disciplines. No prior programming experience is required, and only a modest amount of high school algebra is needed. Here are the key features of this book:

Present fundamentals first.

The book takes a traditional route, first stressing control structures, methods, procedural decomposition, and arrays. Objects are used when appropriate in the early chapters. Students start designing and implementing their own classes in Chapter 8.

Guidance and worked examples help students succeed.

Beginning programmers often ask "How do I start? Now what do I do?" Of course, an activity as complex as programming cannot be reduced to cookbook-style instructions. However, step-by-step guidance is immensely helpful for building confidence and providing an outline for the task at hand. "Problem Solving" sections stress the importance of design and planning. "How To" guides help students with common programming tasks. Additional Worked Examples and Video Examples are available online.

Problem solving strategies are made explicit.

Practical, step-by-step illustrations of techniques help students devise and evaluate solutions to programming problems. Introduced where they are most relevant, these strategies address barriers to success for many students. Strategies included are:

- Algorithm Design (with pseudocode)
- Tracing Objects
- First Do It By Hand (doing sample calculations by hand)
- Flowcharts
- Selecting Test Cases
- Hand-Tracing
- Storyboards
- Solve a Simpler Problem First
- Adapting Algorithms
- Discovering Algorithms by Manipulating Physical Objects
- Patterns for Object Data
- Estimating the Running Time of an Algorithm

Practice makes perfect.

Of course, programming students need to be able to implement nontrivial programs, but they first need to have the confidence that they can succeed. This book contains a substantial number of self-check questions at the end of each section. "Practice It" pointers suggest exercises to try after each section. And additional practice opportunities, including code completion questions and skill-oriented multiple-choice questions, are available online.

A visual approach motivates the reader and eases navigation.

Photographs present visual analogies that explain the nature and behavior of computer concepts. Step-by-step figures illustrate complex program operations. Syntax boxes and example tables present a variety of typical and special cases in a compact format. It is easy to get the "lay of the land" by browsing the visuals, before focusing on the textual material.

© Terraxplorer/iStockphoto.

Visual features help the reader with navigation.

Focus on the essentials while being technically accurate.

An encyclopedic coverage is not helpful for a beginning programmer, but neither is the opposite—reducing the material to a list of simplistic bullet points. In this book, the essentials are presented in digestible chunks, with separate notes that go deeper into good practices or language features when the reader is ready for the additional information. You will not find artificial over-simplifications that give an illusion of knowledge.

Reinforce sound engineering practices.

A multitude of useful tips on software quality and common errors encourage the development of good programming habits. The focus is on test-driven development, encouraging students to test their programs systematically.

Engage with optional science and business exercises.

End-of-chapter exercises are enhanced with problems from scientific and business domains. Designed to engage students, the exercises illustrate the value of programming in applied fields.

New to This Edition

Updated for Java 8

Java 8 introduces many exciting features, and this edition has been updated to take advantage of them. Interfaces can now have default and static methods, and lambda expressions make it easy to provide instances of interfaces with a single method. The sections on interfaces and sorting have been updated to make these innovations optionally available. A new chapter covers the Java 8 stream library and its applications for "big data" processing.

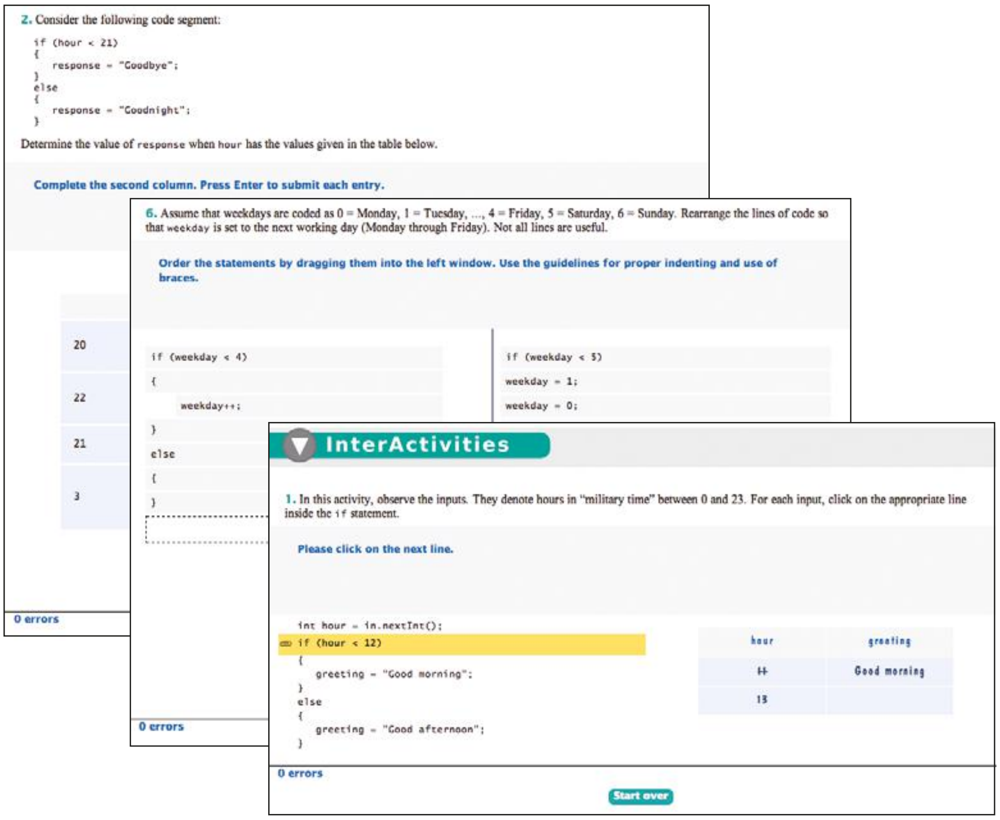
In addition, Java 7 features such as the `try`-with-resources statement are now integrated into the text. Chapter 20 covers the utilities provided by the `Paths` and `Files` classes.

Optional JavaFX Coverage

For those who prefer to use JavaFX instead of Swing, there is a new online resource that covers graphical user-interface programming with JavaFX.

Interactive Learning

Additional interactive content is available that integrates with this text and immerses students in activities designed to foster in-depth learning. Students don't just watch animations and code traces, they work on generating them. The activities provide instant feedback to show students what they did right and where they need to study more. To find out more about how to make this content available in your course, visit `http://wiley.com/go/bjlo2interactivities`.



2. Consider the following code segment:

```
if (hour < 21)
{
   response = "Goodbye";
}
else
{
   response = "Goodnight";
}
```

Determine the value of `response` when `hour` has the values given in the table below.

Complete the second column. Press Enter to submit each entry.

20	
22	
21	
3	

0 errors

6. Assume that weekdays are coded as 0 = Monday, 1 = Tuesday, ..., 4 = Friday, 5 = Saturday, 6 = Sunday. Rearrange the lines of code so that `weekday` is set to the next working day (Monday through Friday). Not all lines are useful.

Order the statements by dragging them into the left window. Use the guidelines for proper indenting and use of braces.

```
if (weekday < 4)
{
    weekday++;
}
else
{
}
```

```
if (weekday < 5)
weekday = 1;
weekday = 0;
```

0 errors

1. In this activity, observe the inputs. They denote hours in "military time" between 0 and 23. For each input, click on the appropriate line inside the `if` statement.

Please click on the next line.

```
int hour = in.nextInt();
if (hour < 12)
{
   greeting = "Good morning";
}
else
{
   greeting = "Good afternoon";
}
```

hour	greeting
11	Good morning
13	

0 errors

"CodeCheck" is an innovative online service that students can use to work on programming problems. You can assign exercises that have already been prepared, and you can easily add your own. Visit `http://codecheck.it` to learn more and to try it out.

A Tour of the Book

This book is intended for a two-semester introduction to programming that may also include algorithms, data structures, and/or applications.

Part A: Fundamentals (Chapters 1–7)

The first seven chapters follow a traditional approach to basic programming concepts. Students learn about control structures, stepwise refinement, and arrays.

Objects are used only for input/output and string processing. Input/output is covered in Chapter 7, but Sections 7.1 and 7.2 can be covered with Chapter 4; in that way, students can practice writing loops that process text files. Chapter 4 also provides an optional introduction to programming drawings that consist of lines, rectangles, and ovals, with an emphasis on reinforcing loops.

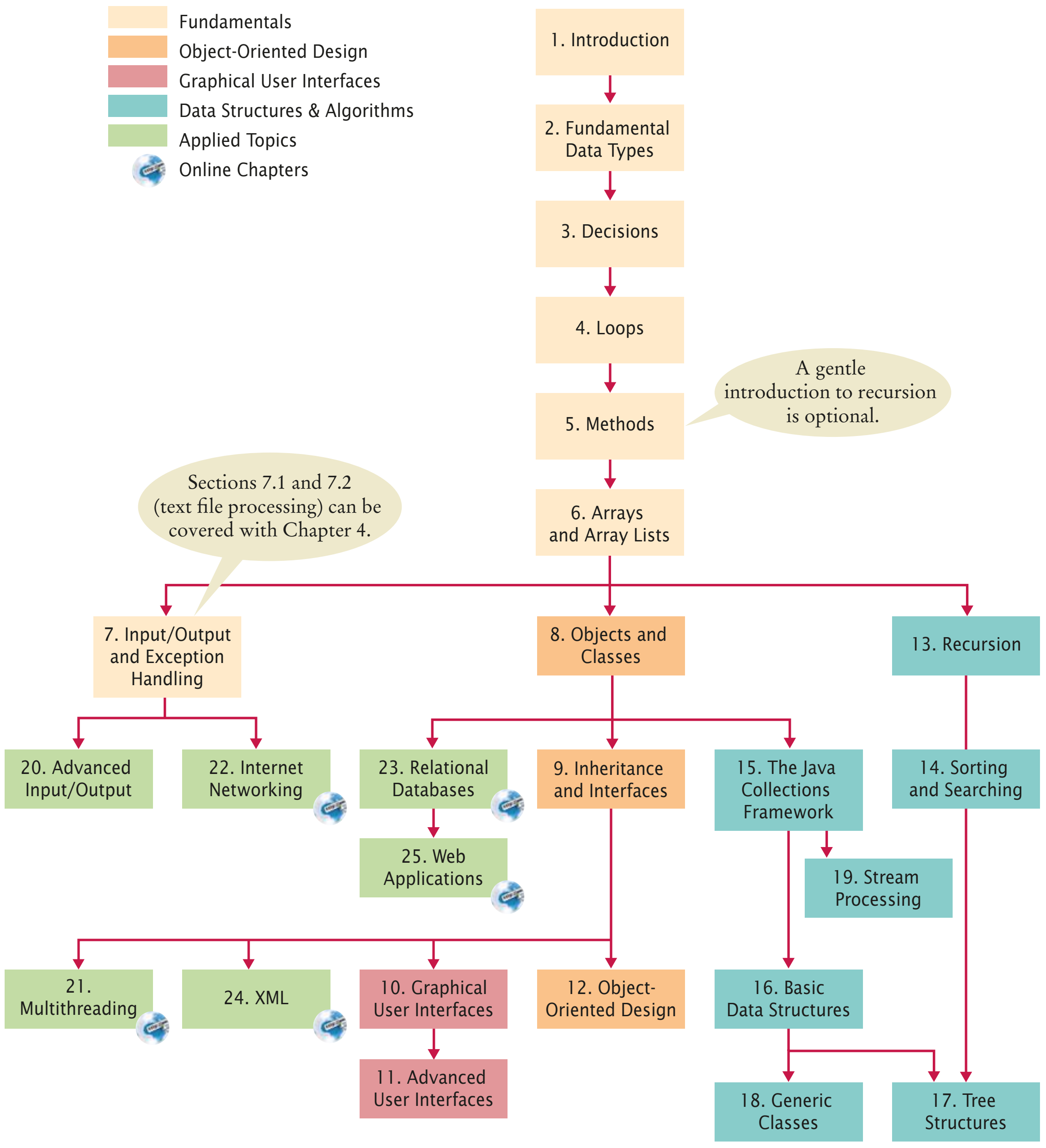

Fundamentals
Object-Oriented Design
Graphical User Interfaces
Data Structures & Algorithms
Applied Topics
Online Chapters
1. Introduction
2. Fundamental Data Types
3. Decisions
4. Loops
5. Methods
A gentle introduction to recursion is optional.
6. Arrays and Array Lists
Sections 7.1 and 7.2 (text file processing) can be covered with Chapter 4.
7. Input/Output and Exception Handling
8. Objects and Classes
13. Recursion
20. Advanced Input/Output
22. Internet Networking
23. Relational Databases
9. Inheritance and Interfaces
15. The Java Collections Framework
14. Sorting and Searching
25. Web Applications
19. Stream Processing
21. Multithreading
24. XML
10. Graphical User Interfaces
12. Object-Oriented Design
16. Basic Data Structures
11. Advanced User Interfaces
18. Generic Classes
17. Tree Structures

Figure 1 Chapter Dependencies

Part B: Object-Oriented Design and Graphics (Chapters 8–12)

After students have gained a solid foundation, they are ready to tackle the implementation of classes in Chapter 8. Chapter 9 covers inheritance and interfaces. A simple methodology for object-oriented design is presented in Chapter 12. Object-oriented design may also be covered immediately after Chapter 9 by omitting the GUI versions of the sample programs. By the end of these chapters, students will be able to implement programs with multiple interacting classes.

Graphical user interfaces are presented in Chapters 10 and 11. The first of these chapters enables students to write programs with buttons, text components, and simple drawings. If you want to go deeper, you will find layout management and additional user-interface components in the second chapter. Online versions of these chapters cover JavaFX instead of Swing.

Part C: Data Structures and Algorithms (Chapters 13–19)

Chapters 13–19 cover algorithms and data structures at a level suitable for beginning students. Recursion, in Chapter 13, starts with simple examples and progresses to meaningful applications that would be difficult to implement iteratively. Chapter 14 covers quadratic sorting algorithms as well as merge sort, with an informal introduction to big-Oh notation. In Chapter 15, the Java Collections Framework is presented from the perspective of a library user, without revealing the implementations of lists and maps. You can cover this chapter anytime after Chapter 8. In Chapters 16 and 17, students learn how to implement linear and tree-based data structures, and how to analyze the efficiency of operations on these data structures. Chapter 18 covers programming with Java generics. Chapter 19 presents the Java 8 stream library.

Part D: Applied Topics (Chapters 20–25)

Chapters 20–25 feature applied topics: binary input/output, concurrent programming, networking, database programming, XML processing, and the development of web applications. Chapters 21–25 are available in electronic form on the Web.

Any subset of these chapters can be incorporated into a custom print version of this text; ask your Wiley sales representative for details.

Appendices

Many instructors find it highly beneficial to require a consistent style for all assignments. If the style guide in Appendix E conflicts with instructor sentiment or local customs, however, it is available in electronic form so that it can be modified. Appendices F–J are available on the Web.

A. The Basic Latin and Latin-1 Subsets of Unicode
B. Java Operator Summary
C. Java Reserved Word Summary
D. The Java Library
E. Java Language Coding Guidelines
F. Tool Summary
G. Number Systems
H. UML Summary
I. Java Syntax Summary
J. HTML Summary

Custom Book and eBook Options

Big Java may be ordered in both custom print and eBook formats. You can order a custom print version that includes your choice of chapters—including those from other Horstmann titles. Visit `customselect.wiley.com` to create your custom order.

Big Java is also available in an electronic eBook format with three key advantages:

- The price is significantly lower than for the printed book.
- The eBook contains all material in the printed book plus the web chapters and worked examples in one easy-to-browse format.
- You can customize the eBook to include your choice of chapters.

The interactive edition of *Big Java* adds even more value by integrating a wealth of interactive exercises into the eBook. See `http://wiley.com/go/bjlo2interactivities` to find out more about this new format.

Please contact your Wiley sales rep for more information about any of these options or check `www.wiley.com/college/horstmann` for available versions.

Web Resources

This book is complemented by a complete suite of online resources. Go to `www.wiley.com/college/horstmann` to visit the online companion sites, which include

- Source code for all example programs in the book and its Worked Examples and Video Examples, plus additional example programs.
- Worked Examples that apply the problem-solving steps in the book to other realistic examples.
- Video Examples in which the author explains the steps he is taking and shows his work as he solves a programming problem.
- Lecture presentation slides (for instructors only).
- Solutions to all review and programming exercises (for instructors only).
- A test bank that focuses on skills, not just terminology (for instructors only). This extensive set of multiple-choice questions can be used with a word processor or imported into a course management system.
- "CodeCheck" assignments that allow students to work on programming problems presented in an innovative online service and receive immediate feedback. Instructors can assign exercises that have already been prepared, or easily add their own. Visit `http://codecheck.it` to learn more.

Pointers in the book describe what students will find on the Web.

VIDEO EXAMPLE 1.1 **Compiling and Running a Program**

See a demonstration of how to compile and run a simple Java program. Go to `wiley.com/go/bjlo2videos` to view Video Example 1.1.

WORKED EXAMPLE 4.1 **Credit Card Processing**

Learn how to use a loop to remove spaces from a credit card number. Go to `wiley.com/go/bjlo2examples` and download Worked Example 4.1.

FULL CODE EXAMPLE

Go to `wiley.com/go/bjlo2code` to download a program that demonstrates variables and assignments.

A Walkthrough of the Learning Aids

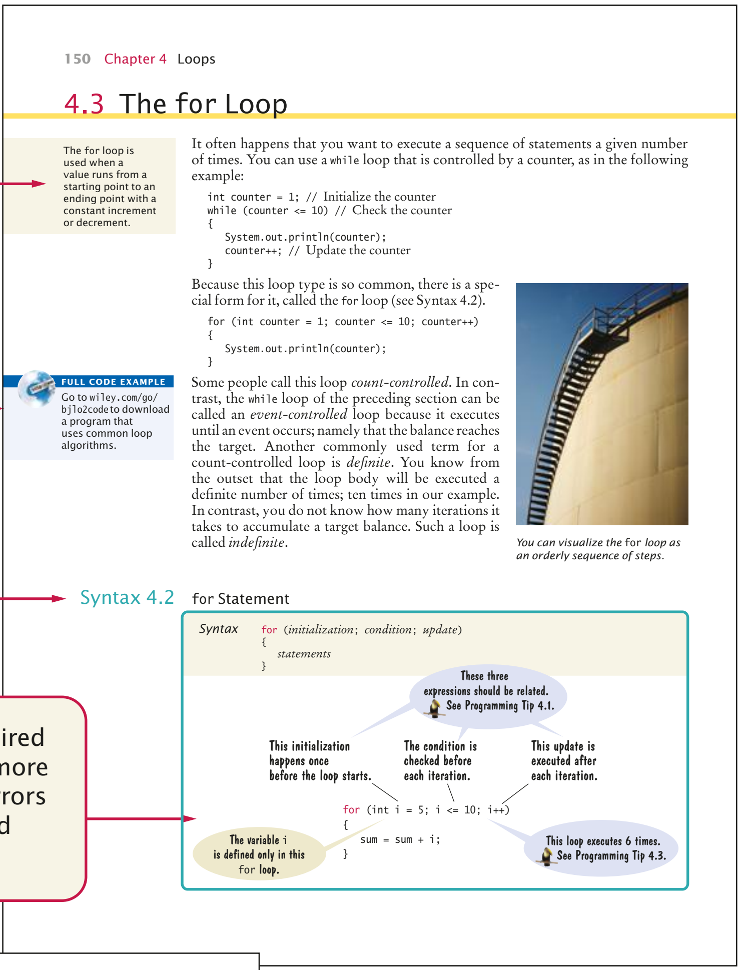
The pedagogical elements in this book work together to focus on and reinforce key concepts and fundamental principles of programming, with additional tips and detail organized to support and deepen these fundamentals. In addition to traditional features, such as chapter objectives and a wealth of exercises, each chapter contains elements geared to today's visual learner.

Throughout each chapter, **margin notes** show where new concepts are introduced and provide an outline of key ideas.

Additional **full code examples** provides complete programs for students to run and modify.

Annotated **syntax boxes** provide a quick, visual overview of new language constructs.

Annotations explain required components and point to more information on common errors or best practices associated with the syntax.



150 Chapter 4 Loops

4.3 The for Loop

The for loop is used when a value runs from a starting point to an ending point with a constant increment or decrement.

It often happens that you want to execute a sequence of statements a given number of times. You can use a `while` loop that is controlled by a counter, as in the following example:

```
int counter = 1; // Initialize the counter
while (counter <= 10) // Check the counter
{
   System.out.println(counter);
   counter++; // Update the counter
}
```

Because this loop type is so common, there is a special form for it, called the `for` loop (see Syntax 4.2).

```
for (int counter = 1; counter <= 10; counter++)
{
   System.out.println(counter);
}
```

FULL CODE EXAMPLE
Go to `wiley.com/go/bjlo2code` to download a program that uses common loop algorithms.

Some people call this loop *count-controlled*. In contrast, the `while` loop of the preceding section can be called an *event-controlled* loop because it executes until an event occurs; namely that the balance reaches the target. Another commonly used term for a count-controlled loop is *definite*. You know from the outset that the loop body will be executed a definite number of times; ten times in our example. In contrast, you do not know how many iterations it takes to accumulate a target balance. Such a loop is called *indefinite*.

You can visualize the for *loop as an orderly sequence of steps.*

Syntax 4.2 for Statement

```
Syntax    for (initialization; condition; update)
          {
             statements
          }
```

```
for (int i = 5; i <= 10; i++)
{
   sum = sum + i;
}
```

These three expressions should be related. See Programming Tip 4.1.

This initialization happens once before the loop starts.

The condition is checked before each iteration.

This update is executed after each iteration.

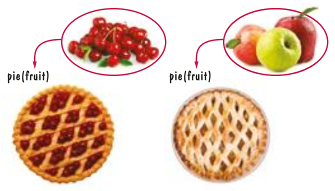
The variable `i` is defined only in this `for` loop.

This loop executes 6 times. See Programming Tip 4.3.

Like a variable in a computer program, a parking space has an identifier and a contents.

Analogies to everyday objects are used to explain the nature and behavior of concepts such as variables, data types, loops, and more.

Memorable photos reinforce analogies and help students remember the concepts.

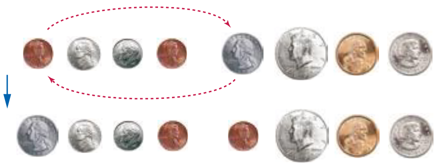


A recipe for a fruit pie may say to use any kind of fruit. Here, "fruit" is an example of a parameter variable. Apples and cherries are examples of arguments.

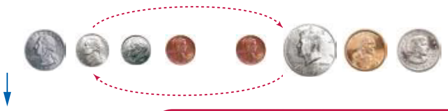
Problem Solving sections teach techniques for generating ideas and evaluating proposed solutions, often using pencil and paper or other artifacts. These sections emphasize that most of the planning and problem solving that makes students successful happens away from the computer.

Now how does that help us with our problem, switching the first and the second half of the array?

Let's put the first coin into place, by swapping it with the fifth coin. However, as Java programmers, we will say that we swap the coins in positions 0 and 4:



Next, we swap the coins in positions 1 and 5:



How To guides give step-by-step guidance for common programming tasks, emphasizing planning and testing. They answer the beginner's question, "Now what do I do?" and integrate key concepts into a problem-solving sequence.

HOW TO 1.1 **Describing an Algorithm with Pseudocode**

This is the first of many "How To" sections in this book that give you step-by-step procedures for carrying out important tasks in developing computer programs.

Before you are ready to write a program in Java, you need to develop an algorithm—a method for arriving at a solution for a particular problem. Describe the algorithm in pseudocode: a sequence of precise steps formulated in English.

For example, consider this problem: You have the choice of buying two cars. One is more fuel efficient than the other, but also more expensive. You know the price and fuel efficiency (in miles per gallon, mpg) of both cars. You plan to keep the car for ten years. Assume a price of $4 per gallon of gas and usage of 15,000 miles per year. You will pay cash for the car and not worry about financing costs. Which car is the better deal?

Step 1 Determine the inputs and outputs.

In our sample problem, we have these inputs:

- **purchase price1** and **fuel efficiency1**, the price and fuel efficiency (in mpg) of the first car
- **purchase price2** and **fuel efficiency2**, the price and fuel efficiency of the second car

We simply want to know which car is the better buy. That is the desired output.

Worked Examples and **Video Examples** apply the steps in the How To to a different example, showing how they can be used to plan, implement, and test a solution to another programming problem.

WORKED EXAMPLE 1.1 **Writing an Algorithm for Tiling a Floor**

This Worked Example shows how to develop an algorithm for laying tile in an alternating pattern of colors.

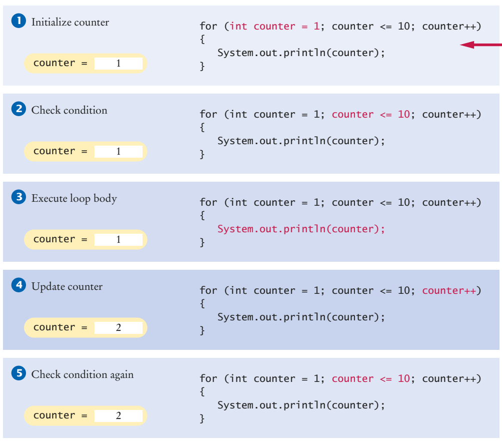
Table 1 Variable Declarations in Java

Variable Name	Comment
`int cans = 6;`	Declares an integer variable and initializes it with 6.
`int total = cans + bottles;`	The initial value need not be a constant. (Of course, `cans` and `bottles` must have been previously declared.)
🚫 `bottles = 1;`	**Error:** The type is missing. This statement is not a declaration but an assignment of a new value to an existing variable—see Section 2.1.4.
🚫 `int bottles = "10";`	**Error:** You cannot initialize a number with a string.
`int bottles;`	Declares an integer variable without initializing it. This can be a cause for errors—see Common Error 2.1 on page 37.
`int cans, bottles;`	Declares two integer variables in a single statement. In this book, we will declare each variable in a separate statement.

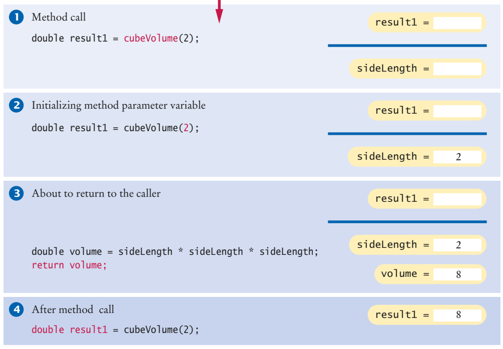
Example tables support beginners with multiple, concrete examples. These tables point out common errors and present another quick reference to the section's topic.

Progressive figures trace code segments to help students visualize the program flow. Color is used consistently to make variables and other elements easily recognizable.

Figure 3 Execution of a for Loop



```
for (int counter = 1; counter <= 10; counter++)
{
   System.out.println(counter);
}
```



```
double result1 = cubeVolume(2);
```

```
double volume = sideLength * sideLength * sideLength;
return volume;
```

Figure 3 Parameter Passing

- The parameter variable sideLength of the cubeVolume method is created when the method is called. 1
- The parameter variable is initialized with the value of the argument that was passed in the call. In our case, sideLength is set to 2. 2
- The method computes the expression sideLength * sideLength * sideLength, which has the value 8. That value is stored in the variable volume. 3
- The method returns. All of its variables are removed. The return value is trans-

Self-check exercises at the end of each section are designed to make students think through the new material—and can spark discussion in lecture.

SELF CHECK

11. Write the for loop of the InvestmentTable.java program as a while loop.
12. How many numbers does this loop print?
```
for (int n = 10; n >= 0; n--)
{
   System.out.println(n);
}
```
13. Write a for loop that prints all even numbers between 10 and 20 (inclusive).
14. Write a for loop that computes the sum of the integers from 1 to n.
15. How would you modify the for loop of the InvestmentTable.java program to print all balances until the investment has doubled?

Practice It Now you can try these exercises at the end of the chapter: R4.7, R4.13, E4.8, E4.16.

Optional **science and business exercises** engage students with realistic applications of Java.

■■ Science P6.13 Sounds can be represented by an array of "sample values" that describe the intensity of the sound at a point in time. The program ch06/sound/SoundEffect.java reads a sound file (in WAV format), calls a method process for processing the sample values, and saves the sound file. Your task is to implement the process method by introducing an echo. For each

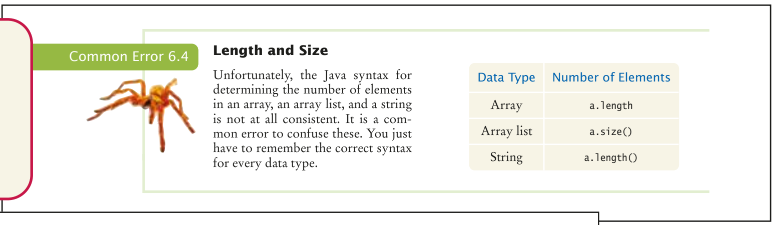
■■ Business P9.6 Implement a superclass Appointment and subclasses Onetime, Daily, and Monthly. An appointment has a description (for example, "see the dentist") and a date and time. Write a method occursOn(int year, int month, int day) that checks whether the appointment occurs on that date. For example, for a monthly appointment, you must check whether the day of the month matches. Then fill an array of Appointment objects with a mixture of appointments. Have the user enter a date and print out all appointments that occur on that date.

sec01/DoubleInvestment.java

```
/**
   This program computes the time required to double an investment.
*/
public class DoubleInvestment
{
   public static void main(String[] args)
   {
      final double RATE = 5;
      final double INITIAL_BALANCE = 10000;
      final double TARGET = 2 * INITIAL_BALANCE;

      double balance = INITIAL_BALANCE;
      int year = 0;

      // Count the years required for the investment to double

      while (balance < TARGET)
      {
         year++;
         double interest = balance * RATE / 100;
         balance = balance + interest;
      }

      System.out.println("The investment doubled after "
         + year + " years.");
   }
}
```

Program listings are carefully designed for easy reading, going well beyond simple color coding. Methods are set off by a subtle outline.

Common Errors describe the kinds of errors that students often make, with an explanation of why the errors occur, and what to do about them.



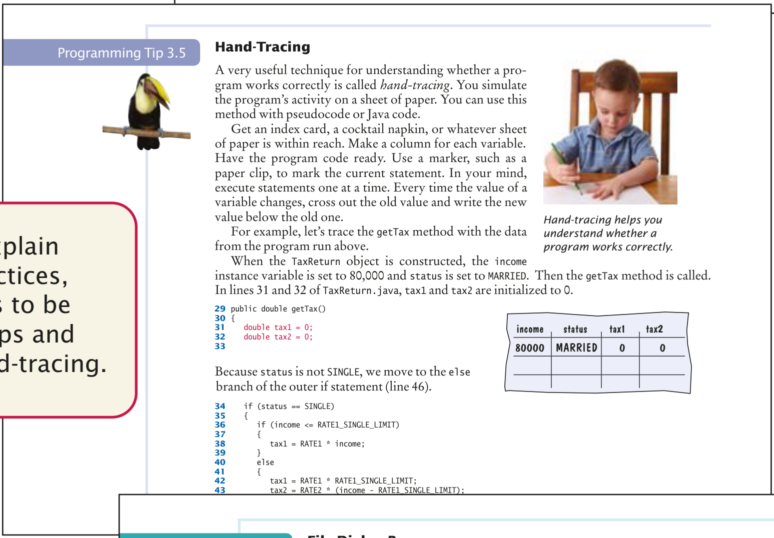
Common Error 6.4

Length and Size

Unfortunately, the Java syntax for determining the number of elements in an array, an array list, and a string is not at all consistent. It is a common error to confuse these. You just have to remember the correct syntax for every data type.

Data Type	Number of Elements
Array	`a.length`
Array list	`a.size()`
String	`a.length()`

Programming Tips explain good programming practices, and encourage students to be more productive with tips and techniques such as hand-tracing.



Programming Tip 3.5

Hand-Tracing

A very useful technique for understanding whether a program works correctly is called *hand-tracing*. You simulate the program's activity on a sheet of paper. You can use this method with pseudocode or Java code.

Get an index card, a cocktail napkin, or whatever sheet of paper is within reach. Make a column for each variable. Have the program code ready. Use a marker, such as a paper clip, to mark the current statement. In your mind, execute statements one at a time. Every time the value of a variable changes, cross out the old value and write the new value below the old one.

Hand-tracing helps you understand whether a program works correctly.

For example, let's trace the `getTax` method with the data from the program run above.

When the `TaxReturn` object is constructed, the `income` instance variable is set to 80,000 and `status` is set to `MARRIED`. Then the `getTax` method is called. In lines 31 and 32 of `TaxReturn.java`, `tax1` and `tax2` are initialized to `0`.

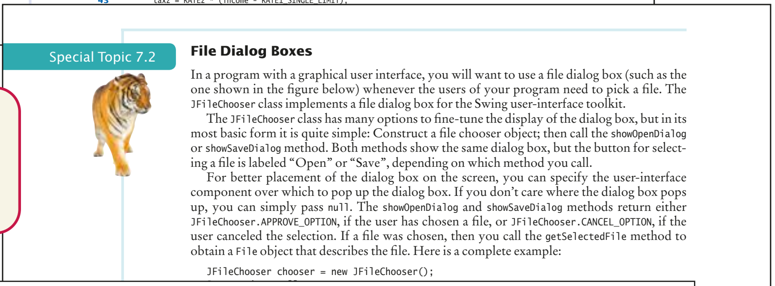
```
29 public double getTax()
30 {
31    double tax1 = 0;
32    double tax2 = 0;
33
```

income	status	tax1	tax2
80000	MARRIED	0	0

Because `status` is not `SINGLE`, we move to the `else` branch of the outer if statement (line 46).

```
34    if (status == SINGLE)
35    {
36       if (income <= RATE1_SINGLE_LIMIT)
37       {
38          tax1 = RATE1 * income;
39       }
40       else
41       {
42          tax1 = RATE1 * RATE1_SINGLE_LIMIT;
43          tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
```

Special Topics present optional topics and provide additional explanation of others.



Special Topic 7.2

File Dialog Boxes

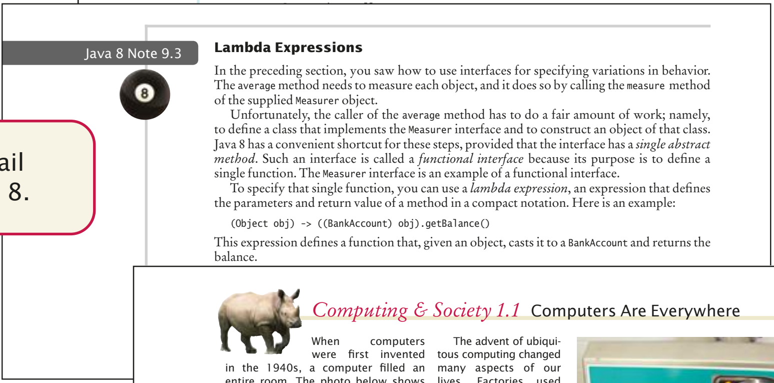
In a program with a graphical user interface, you will want to use a file dialog box (such as the one shown in the figure below) whenever the users of your program need to pick a file. The `JFileChooser` class implements a file dialog box for the Swing user-interface toolkit.

The `JFileChooser` class has many options to fine-tune the display of the dialog box, but in its most basic form it is quite simple: Construct a file chooser object; then call the `showOpenDialog` or `showSaveDialog` method. Both methods show the same dialog box, but the button for selecting a file is labeled "Open" or "Save", depending on which method you call.

For better placement of the dialog box on the screen, you can specify the user-interface component over which to pop up the dialog box. If you don't care where the dialog box pops up, you can simply pass `null`. The `showOpenDialog` and `showSaveDialog` methods return either `JFileChooser.APPROVE_OPTION`, if the user has chosen a file, or `JFileChooser.CANCEL_OPTION`, if the user canceled the selection. If a file was chosen, then you call the `getSelectedFile` method to obtain a `File` object that describes the file. Here is a complete example:

```
JFileChooser chooser = new JFileChooser();
```

Java 8 Notes provide detail about new features in Java 8.



Java 8 Note 9.3

Lambda Expressions

In the preceding section, you saw how to use interfaces for specifying variations in behavior. The `average` method needs to measure each object, and it does so by calling the `measure` method of the supplied `Measurer` object.

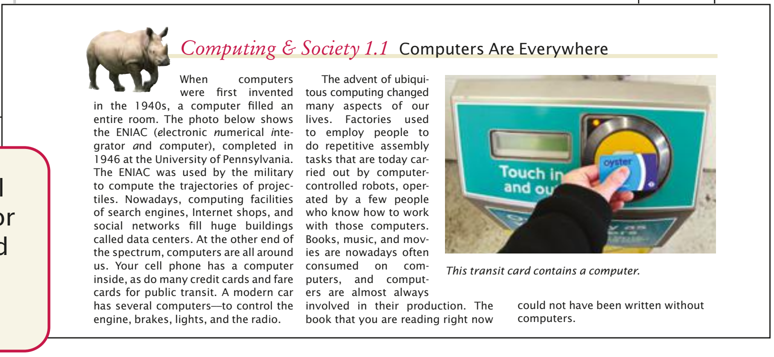
Unfortunately, the caller of the `average` method has to do a fair amount of work; namely, to define a class that implements the `Measurer` interface and to construct an object of that class. Java 8 has a convenient shortcut for these steps, provided that the interface has a *single abstract method*. Such an interface is called a *functional interface* because its purpose is to define a single function. The `Measurer` interface is an example of a functional interface.

To specify that single function, you can use a *lambda expression*, an expression that defines the parameters and return value of a method in a compact notation. Here is an example:

```
(Object obj) -> ((BankAccount) obj).getBalance()
```

This expression defines a function that, given an object, casts it to a `BankAccount` and returns the balance.

Computing & Society presents social and historical topics on computing—for interest and to fulfill the "historical and social context" requirements of the ACM/IEEE curriculum guidelines.



Computing & Society 1.1 Computers Are Everywhere

When computers were first invented in the 1940s, a computer filled an entire room. The photo below shows the ENIAC (*e*lectronic *n*umerical *i*ntegrator *a*nd *c*omputer), completed in 1946 at the University of Pennsylvania. The ENIAC was used by the military to compute the trajectories of projectiles. Nowadays, computing facilities of search engines, Internet shops, and social networks fill huge buildings called data centers. At the other end of the spectrum, computers are all around us. Your cell phone has a computer inside, as do many credit cards and fare cards for public transit. A modern car has several computers—to control the engine, brakes, lights, and the radio.

The advent of ubiquitous computing changed many aspects of our lives. Factories used to employ people to do repetitive assembly tasks that are today carried out by computer-controlled robots, operated by a few people who know how to work with those computers. Books, music, and movies are nowadays often consumed on computers, and computers are almost always involved in their production. The book that you are reading right now could not have been written without computers.

This transit card contains a computer.

Acknowledgments

Many thanks to Don Fowley, Graig Donini, Brad Franklin, Dan Sayre, Anna Pham, Laura Abrams, and Billy Ray at John Wiley & Sons for their help with this project. An especially deep acknowledgment and thanks goes to Cindy Johnson for her hard work, sound judgment, and amazing attention to detail.

I am grateful to Jose Cordova, *University of Louisiana at Monroe*, Suzanne Dietrich, *Arizona State University, West Campus*, Byron Hoy, *Stockton University*, Brent Wilson, *George Fox University*, and David Woolbright, *Columbus State University*, for their contributions to the supplemental material.

Every new edition builds on the suggestions and experiences of new and prior reviewers, contributors, and users. Many thanks to the individuals who provided feedback, reviewed the manuscript, made valuable suggestions and contributions, and brought errors and omissions to my attention. They include:

Lynn Aaron, *SUNY Rockland Community College*
Karen Arlien, *Bismarck State College*
Jay Asundi, *University of Texas, Dallas*
Eugene Backlin, *DePaul University*
William C. Barge, *Trine University*
Bruce J. Barton, *Suffolk County Community College*
Sanjiv K. Bhatia, *University of Missouri, St. Louis*
Anna Bieszczad, *California State University, Channel Islands*
Jackie Bird, *Northwestern University*
Eric Bishop, *Northland Pioneer College*
Paul Bladek, *Edmonds Community College*
Paul Logasa Bogen II, *Texas A&M University*
Irene Bruno, *George Mason University*
Paolo Bucci, *Ohio State University*
Joe Burgin, *College of Southern Maryland*
Robert P. Burton, *Brigham Young University*
Leonello Calabresi, *University of Maryland University College*
Martine Ceberio, *University of Texas, El Paso*
Uday Chakraborty, *University of Missouri, St. Louis*
Suchindran Chatterjee, *Arizona State University*
Xuemin Chen, *Texas Southern University*
Haiyan Cheng, *Willamette University*
Chakib Chraibi, *Barry University*
Ta-Tao Chuang, *Gonzaga University*
Vincent Cicirello, *Richard Stockton College*
Mark Clement, *Brigham Young University*
Gerald Cohen, *St. Joseph's College*
Ralph Conrad, *San Antonio College*
Dave Cook, *Stephen F. Austin State University*
Rebecca Crellin, *Community College of Allegheny County*
Leslie Damon, *Vermont Technical College*
Geoffrey D. Decker, *Northern Illinois University*
Khaled Deeb, *Barry University, School of Adult and Continuing Education*
Akshaye Dhawan, *Ursinus College*
Julius Dichter, *University of Bridgeport*
Mike Domaratzki, *University of Manitoba*
Philip Dorin, *Loyola Marymount University*
Anthony J. Dos Reis, *SUNY New Paltz*
Elizabeth Drake, *Santa Fe College*
Tom Duffy, *Norwalk Community College*
Michael Eckmann, *Skidmore College*
Sander Eller, *California State Polytechnic University, Pomona*
Amita Engineer, *Valencia Community College*
Dave Evans, *Pasadena Community College*
James Factor, *Alverno College*
Chris Fietkiewicz, *Case Western Reserve University*
Terrell Foty, *Portland Community College*
Valerie Frear, *Daytona State College*
Zhenguang Gao, *Framingham State University*
Ryan Garlick, *University of North Texas*
Aaron Garrett, *Jacksonville State University*
Stephen Gilbert, *Orange Coast College*
Rick Giles, *Acadia University*
Peter van der Goes, *Rose State College*
Billie Goldstein, *Temple University*
Michael Gourley, *University of Central Oklahoma*
Grigoriy Grinberg, *Montgomery College*
Linwu Gu, *Indiana University*
Sylvain Guinepain, *University of Oklahoma, Norman*
Bruce Haft, *Glendale Community College*
Nancy Harris, *James Madison University*
Allan M. Hart, *Minnesota State University, Mankato*

Ric Heishman, *George Mason University*
Guy Helmer, *Iowa State University*
Katherin Herbert, *Montclair State University*
Rodney Hoffman, *Occidental College*
May Hou, *Norfolk State University*
John Houlihan, *Loyola University*
Andree Jacobson, *University of New Mexico*
Eric Jiang, *University of San Diego*
Christopher M. Johnson, *Guilford College*
Jonathan Kapleau, *New Jersey Institute of Technology*
Amitava Karmaker, *University of Wisconsin, Stout*
Rajkumar Kempaiah, *College of Mount Saint Vincent*
Mugdha Khaladkar, *New Jersey Institute of Technology*
Richard Kick, *Newbury Park High School*
Julie King, *Sullivan University, Lexington*
Samuel Kohn, *Touro College*
April Kontostathis, *Ursinus College*
Ron Krawitz, *DeVry University*
Nat Kumaresan, *Georgia Perimeter College*
Debbie Lamprecht, *Texas Tech University*
Jian Lin, *Eastern Connecticut State University*
Hunter Lloyd, *Montana State University*
Cheng Luo, *Coppin State University*
Kelvin Lwin, *University of California, Merced*
Frank Malinowski, *Dalton College*
John S. Mallozzi, *Iona College*
Khaled Mansour, *Washtenaw Community College*
Kenneth Martin, *University of North Florida*
Deborah Mathews, *J. Sargeant Reynolds Community College*
Louis Mazzucco, *State University of New York at Cobleskill and Excelsior College*
Drew McDermott, *Yale University*
Patricia McDermott-Wells, *Florida International University*
Hugh McGuire, *Grand Valley State University*
Michael L. Mick, *Purdue University, Calumet*
Jeanne Milostan, *University of California, Merced*
Sandeep Mitra, *SUNY Brockport*
Michel Mitri, *James Madison University*
Kenrick Mock, *University of Alaska Anchorage*
Namdar Mogharreban, *Southern Illinois University*
Jose-Arturo Mora-Soto, *University of Madrid*
Shamsi Moussavi, *Massbay Community College*
Nannette Napier, *Georgia Gwinnett College*
Tony Tuan Nguyen, *De Anza College*
Michael Ondrasek, *Wright State University*
K. Palaniappan, *University of Missouri*
James Papademas, *Oakton Community College*
Gary Parker, *Connecticut College*
Jody Paul, *Metropolitan State College of Denver*
Mark Pendergast, *Florida Gulf Coast University*
James T. Pepe, *Bentley University*
Jeff Pittges, *Radford University*
Tom Plunkett, *Virginia Tech*
Linda L. Preece, *Southern Illinois University*
Vijay Ramachandran, *Colgate University*
Craig Reinhart, *California Lutheran University*
Jonathan Robinson, *Touro College*
Chaman Lal Sabharwal, *Missouri University of Science & Technology*
Katherine Salch, *Illinois Central College*
Namita Sarawagi, *Rhode Island College*
Ben Schafer, *University of Northern Iowa*
Walter Schilling, *Milwaukee School of Engineering*
Jeffrey Paul Scott, *Blackhawk Technical College*
Amon Seagull, *NOVA Southeastern University*
Brent Seales, *University of Kentucky*
Linda Seiter, *John Carroll University*
Kevin Seppi, *Brigham Young University*
Ricky J. Sethi, *UCLA, USC ISI, and DeVry University*
Narasimha Shashidhar, *Sam Houston State University*
Ali Shaykhian, *Florida Institute of Technology*
Lal Shimpi, *Saint Augustine's College*
Victor Shtern, *Boston University*
Rahul Simha, *George Washington University*
Jeff Six, *University of Delaware*
Donald W. Smith, *Columbia College*
Derek Snow, *University of Southern Alabama*
Peter Spoerri, *Fairfield University*
David R. Stampf, *Suffolk County Community College*
Peter Stanchev, *Kettering University*
Ryan Stansifer, *Florida Institute of Technology*
Stu Steiner, *Eastern Washington University*
Robert Strader, *Stephen F. Austin State University*
David Stucki, *Otterbein University*
Ashok Subramanian, *University of Missouri, St Louis*
Jeremy Suing, *University of Nebraska, Lincoln*
Dave Sullivan, *Boston University*
Vaidy Sunderam, *Emory University*
Hong Sung, *University of Central Oklahoma*
Monica Sweat, *Georgia Tech University*
Joseph Szurek, *University of Pittsburgh, Greensburg*
Jack Tan, *University of Wisconsin*
Cynthia Tanner, *West Virginia University*
Russell Tessier, *University of Massachusetts, Amherst*
Krishnaprasad Thirunarayan, *Wright State University*

Mark Thomas, *University of Cincinnati Blue Ash*
Megan Thomas, *California State University, Stanislaus*
Timothy Urness, *Drake University*
Eliana Valenzuela-Andrade, *University of Puerto Rico at Arecibo*
Tammy VanDeGrift, *University of Portland*
Philip Ventura, *Broward College*
David R. Vineyard, *Kettering University*
Qi Wang, *Northwest Vista College*
Jonathan Weissman, *Finger Lakes Community College*
Reginald White, *Black Hawk Community College*
Ying Xie, *Kennesaw State University*
Arthur Yanushka, *Christian Brothers University*
Chen Ye, *University of Illinois, Chicago*
Wook-Sung Yoo, *Fairfield University*
Steve Zale, *Middlesex County College*
Bahram Zartoshty, *California State University, Northridge*
Frank Zeng, *Indiana Wesleyan University*
Hairong Zhao, *Purdue University Calumet*
Stephen Zilora, *Rochester Institute of Technology*

And a special thank you to our class testers:

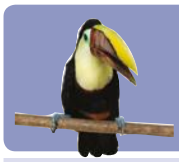
Eugene Backlin and the students of DePaul University, Loop
Debra M. Duke and the students of J. Sargeant Reynolds Community College
Gerald Gordon and the students of DePaul University, Loop
Mike Gourley and the students of the University of Central Oklahoma
Mohammad Morovati and the students of the College of DuPage
Mutsumi Nakamura and the students of Arizona State University
George Novacky and the students of the University of Pittsburgh
Darrin Rothe and the students of the Milwaukee School of Engineering
Paige Rutner and the students of Georgia Southern University
Narasimha Shashidhar and the students of Sam Houston State University
Mark Sherriff and the students of the University of Virginia
Frank Zeng and the students of Indiana Wesleyan University

CONTENTS

ALPHABETICAL LIST OF SYNTAX BOXES

CHAPTER	Common Errors		How Tos and Worked Examples	
1 Introduction	Omitting Semicolons	13	Compiling and Running a Program	Video
	Misspelling Words	15	Describing an Algorithm with Pseudocode	19
			Writing an Algorithm for Tiling a Floor	🌐
			Dividing Household Expenses	Video
2 Fundamental Data Types	Using Undeclared or Uninitialized Variables	39	Using Integer Division	Video
	Overflow	40	Carrying out Computations	56
	Roundoff Errors	40	Computing the Cost of Stamps	🌐
	Unintended Integer Division	48	Computing Travel Time	🌐
	Unbalanced Parentheses	48	Computing Distances on Earth	Video
3 Decisions	A Semicolon After the `if` Condition	88	Implementing an `if` Statement	95
	Exact Comparison of Floating-Point Numbers	93	Extracting the Middle	🌐
	Using == to Compare Strings	94	Computing the Plural of an English Word	Video
	The Dangling `else` Problem	106	The Genetic Code	Video
	Combining Multiple Relational Operators	115		
	Confusing && and `||` Conditions	116		
4 Loops	Don't Think "Are We There Yet?"	146	Evaluating a Cell Phone Plan	Video
	Infinite Loops	147	Writing a Loop	171
	Off-by-One Errors	147	Credit Card Processing	🌐
			Manipulating the Pixels in an Image	🌐
			Drawing a Spiral	Video
5 Methods	Trying to Modify Arguments	219	Implementing a Method	222
	Missing Return Value	222	Generating Random Passwords	🌐
			Calculating a Course Grade	🌐
			Debugging	Video
			Thinking Recursively	243
			Fully Justified Text	Video

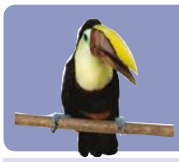
🌐 Available online at www.wiley.com/college/horstmann.



Programming Tips		Special Topics and Java 8 Notes		Computing & Society	
Backup Copies	10			Computers Are Everywhere	5
Choose Descriptive Variable Names	40	Numeric Types in Java	41	The Pentium Floating-Point Bug	50
Do Not Use Magic Numbers	41	Big Numbers	42	International Alphabets and Unicode	68
Spaces in Expressions	49	⑧ Avoiding Negative Remainders	49		
Use the API Documentation	55	Combining Assignment and Arithmetic	49		
		Instance Methods and Static Methods	66		
		Using Dialog Boxes for Input and Output	67		
Brace Layout	88	The Conditional Operator	89	The Denver Airport Luggage System	97
Always Use Braces	88	Lexicographic Ordering of Strings	94	Artificial Intelligence	121
Tabs	89	The `switch` Statement	101		
Avoid Duplication in Branches	90	Enumeration Types	107		
Hand-Tracing	105	Logging	112		
Make a Schedule and Make Time for Unexpected Problems	111	Short-Circuit Evaluation of Boolean Operators	116		
		De Morgan's Law	117		
Use for Loops for Their Intended Purpose Only	157	The Loop-and-a-Half Problem and the break Statement	162	The First Bug	148
Choose Loop Bounds That Match Your Task	157	Redirection of Input and Output	163	Digital Piracy	188
Count Iterations	158	Drawing Graphical Shapes	186		
Flowcharts for Loops	159				
Method Comments	217			Personal Computing	228
Do Not Modify Parameter Variables	219				
Keep Methods Short	234				
Tracing Methods	234				
Stubs	235				
Using a Debugger	239				

Available online at `www.wiley.com/college/horstmann`.

🌐 Available online at `www.wiley.com/college/horstmann`.



Available online at `www.wiley.com/college/horstmann`.

CHAPTER	Common Errors		How Tos and Worked Examples	
10 Graphical User Interfaces	Modifying Parameter Types in the Implementing Method	506	Drawing Graphical Shapes	525
	Forgetting to Attach a Listener	506	Coding a Bar Chart Creator	🌐
	Forgetting to Repaint	524	Solving Crossword Puzzles	Video
	By Default, Components Have Zero Width and Height	525		
11 Advanced User Interfaces			Laying Out a User Interface	546
			Programming a Working Calculator	🌐
			Adding Mouse and Keyboard Support to the Bar Chart Creator	🌐
			Designing a Baby Naming Program	Video
12 Object-Oriented Design			Using CRC Cards and UML Diagrams in Program Design	586
			Simulating an Automatic Teller Machine	🌐
13 Recursion	Infinite Recursion	612	Thinking Recursively	613
	Tracing Through Recursive Methods	612	Finding Files	🌐
			Towers of Hanoi	🌐
14 Sorting and Searching	The compareTo Method Can Return Any Integer, Not Just −1, 0, and 1	680	Enhancing the Insertion Sort Algorithm	🌐
15 The Java Collections Framework			Choosing a Collection	708
			Word Frequency	🌐
			Simulating a Queue of Waiting Customers	🌐
			Building a Table of Contents	Video

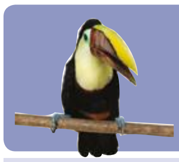
🌐 Available online at www.wiley.com/college/horstmann.

Programming Tips		Special Topics and Java 8 Notes		Computing & Society	
Don’t Use a Frame as a Listener	506	Adding the `main` Method to the Frame Class Local Inner Classes Anonymous Inner Classes ● Lambda Expressions for Event Handling	498 507 508 509		
Use a GUI Builder	548	Keyboard Events Event Adapters	567 568		
Make Parallel Arrays into Arrays of Objects	588	Attributes and Methods in UML Diagrams Multiplicities Aggregation, Association, and Composition	586 587 587	Databases and Privacy	600
				The Limits of Computation	626
		Oh, Omega, and Theta Insertion Sort The Quicksort Algorithm The `Comparator` Interface ● Comparators with Lambda Expressions	658 659 666 680 681	The First Programmer	672
Use Interface References to Manipulate Data Structures	705	● Updating Map Entries Hash Functions Reverse Polish Notation	708 710 723	Standardization	700

Available online at `www.wiley.com/college/horstmann`.

CHAPTER	Common Errors		How Tos and Worked Examples	
16 Basic Data Structures			Implementing a Doubly-Linked List	🌐
17 Tree Structures			Building a Huffman Tree Implementing a Red-Black Tree	🌐 🌐
18 Generic Classes	Genericity and Inheritance The Array Store Exception Using Generic Types in a Static Context	847 847 852	Making a Generic Binary Search Tree Class	🌐
19 Stream Processing	Don't Use a Terminated Stream `Optional` Results Without Values Don't Apply Mutations in Parallel Stream Operations	868 875 877	Working with Streams Word Properties A Movie Database	885 🌐 🌐
20 Advanced Input/Output	Negative byte Values	903	Choosing a File Format	912
21 Multithreading (WEB ONLY) 🌐	Calling await Without Calling `signalAll` Calling `signalAll` Without Locking the Object	🌐 🌐		
22 Internet Networking (WEB ONLY) 🌐			Designing Client/Server Programs	🌐
23 Relational Databases (WEB ONLY) 🌐	Joining Tables Without Specifying a Link Condition Constructing Queries from Arbitrary Strings	🌐 🌐	Programming a Bank Database	🌐
24 XML (WEB ONLY) 🌐	XML Elements Describe Objects, Not Classes	🌐	Designing an XML Document Format Writing an XML Document Writing a DTD	🌐 🌐 🌐
25 Web Applications (WEB ONLY) 🌐			Designing a Managed Bean	🌐

🌐 Available online at `www.wiley.com/college/horstmann`.

 Programming Tips	Special Topics and Java 8 Notes	Computing & Society
	Static Classes 750 Open Addressing 769	
	Wildcard Types 848 Reflection 852	
One Stream Operation Per Line 865 Keep Lambda Expressions Short 870	Infinite Streams 865 Method and Constructor References 871 Higher-Order Functions 872 Higher-Order Functions and Comparators 873	
Use the Runnable Interface 🌐 Check for Thread Interruptions in the run Method of a Thread 🌐	Thread Pools 🌐 Object Locks and Synchronized Methods 🌐 The Java Memory Model 🌐	
Use High-Level Libraries 🌐		
Stick with the Standard 🌐 Avoid Unnecessary Data Replication 🌐 Don't Replicate Columns in a Table 🌐 Don't Hardwire Database Connection Parameters into Your Program 🌐 Let the Database Do the Work 🌐	Primary Keys and Indexes 🌐 Transactions 🌐 Object-Relational Mapping 🌐	
Prefer XML Elements over Attributes 🌐 Avoid Children with Mixed Elements and Text 🌐	Grammars, Parsers, and Compilers 🌐 Schema Languages 🌐 Other XML Technologies 🌐	
	Session State and Cookies 🌐 AJAX 🌐	

🌐 Available online at www.wiley.com/college/horstmann.

CHAPTER 1

INTRODUCTION

© JanPietruszka/iStockphoto.

CHAPTER GOALS

To learn about computers and programming

To compile and run your first Java program

To recognize compile-time and run-time errors

To describe an algorithm with pseudocode

CHAPTER CONTENTS

© JanPietruszka/iStockphoto.

Just as you gather tools, study a project, and make a plan for tackling it, in this chapter you will gather up the basics you need to start learning to program. After a brief introduction to computer hardware, software, and programming in general, you will learn how to write and run your first Java program. You will also learn how to diagnose and fix programming errors, and how to use pseudocode to describe an algorithm—a step-by-step description of how to solve a problem—as you plan your computer programs.

1.1 Computer Programs

Computers execute very basic instructions in rapid succession.

You have probably used a computer for work or fun. Many people use computers for everyday tasks such as electronic banking or writing a term paper. Computers are good for such tasks. They can handle repetitive chores, such as totaling up numbers or placing words on a page, without getting bored or exhausted.

The flexibility of a computer is quite an amazing phenomenon. The same machine can balance your checkbook, lay out your term paper, and play a game. In contrast, other machines carry out a much narrower range of tasks; a car drives and a toaster toasts. Computers can carry out a wide range of tasks because they execute different programs, each of which directs the computer to work on a specific task.

A computer program is a sequence of instructions and decisions.

The computer itself is a machine that stores data (numbers, words, pictures), interacts with devices (the monitor, the sound system, the printer), and executes programs. A **computer program** tells a computer, in minute detail, the sequence of steps that are needed to fulfill a task. The physical computer and peripheral devices are collectively called the **hardware**. The programs the computer executes are called the **software**.

Today's computer programs are so sophisticated that it is hard to believe that they are composed of extremely primitive instructions. A typical instruction may be one of the following:

- Put a red dot at a given screen position.
- Add up two numbers.
- If this value is negative, continue the program at a certain instruction.

The computer user has the illusion of smooth interaction because a program contains a huge number of such instructions, and because the computer can execute them at great speed.

Programming is the act of designing and implementing computer programs.

The act of designing and implementing computer programs is called **programming**. In this book, you will learn how to program a computer—that is, how to direct the computer to execute tasks.

To write a computer game with motion and sound effects or a word processor that supports fancy fonts and pictures is a complex task that requires a team of many highly-skilled programmers. Your first programming efforts will be more mundane. The concepts and skills you learn in this book form an important foundation, and you should not be disappointed if your first programs do not rival the sophisticated software that is familiar to you. Actually, you will find that there is an immense thrill even in simple programming tasks. It is an amazing experience to see the computer precisely and quickly carry out a task that would take you hours of drudgery, to make small changes in a program that lead to immediate improvements, and to see the computer become an extension of your mental powers.

SELF CHECK

1. What is required to play music on a computer?
2. Why is a CD player less flexible than a computer?
3. What does a computer user need to know about programming in order to play a video game?

1.2 The Anatomy of a Computer

To understand the programming process, you need to have a rudimentary understanding of the building blocks that make up a computer. We will look at a personal computer. Larger computers have faster, larger, or more powerful components, but they have fundamentally the same design.

At the heart of the computer lies the **central processing unit (CPU)** (see Figure 1). The inside wiring of the CPU is enormously complicated. For example, the Intel Core processor (a popular CPU for personal computers at the time of this writing) is composed of several hundred million structural elements, called *transistors*.

The central processing unit (CPU) performs program control and data processing.

The CPU performs program control and data processing. That is, the CPU locates and executes the program instructions; it carries out arithmetic operations such as addition, subtraction, multiplication, and division; it fetches data from external memory or devices and places processed data into storage.

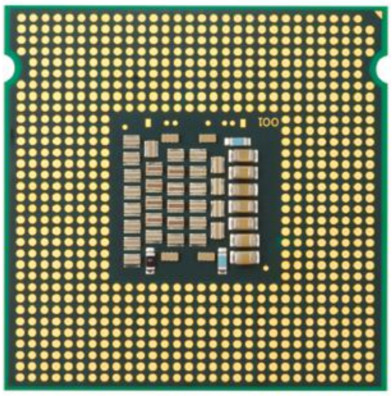

© Amorphis/iStockphoto.

Figure 1 Central Processing Unit

Storage devices include memory and secondary storage.

There are two kinds of storage. Primary storage, or memory, is made from electronic circuits that can store data, provided they are supplied with electric power. **Secondary storage**, usually a **hard disk** (see Figure 2)

PhotoDisc, Inc./Getty Images, Inc.

Figure 2 A Hard Disk

or a solid-state drive, provides slower and less expensive storage that persists without electricity. A hard disk consists of rotating platters, which are coated with a magnetic material. A solid-state drive uses electronic components that can retain information without power, and without moving parts.

To interact with a human user, a computer requires peripheral devices. The computer transmits information (called *output*) to the user through a display screen, speakers, and printers. The user can enter information (called *input*) for the computer by using a keyboard or a pointing device such as a mouse.

Some computers are self-contained units, whereas others are interconnected through **networks**. Through the network cabling, the computer can read data and programs from central storage locations or send data to other computers. To the user of a networked computer, it may not even be obvious which data reside on the computer itself and which are transmitted through the network.

Figure 3 gives a schematic overview of the architecture of a personal computer. Program instructions and data (such as text, numbers, audio, or video) reside in secondary storage or elsewhere on the network. When a program is started, its instructions are brought into memory, where the CPU can read them. The CPU reads and executes one instruction at a time. As directed by these instructions, the CPU reads data, modifies it, and writes it back to memory or secondary storage. Some program instructions will cause the CPU to place dots on the display screen or printer or to vibrate the speaker. As these actions happen many times over and at great speed, the human user will perceive images and sound. Some program instructions read user input from the keyboard, mouse, touch sensor, or microphone. The program analyzes the nature of these inputs and then executes the next appropriate instruction.

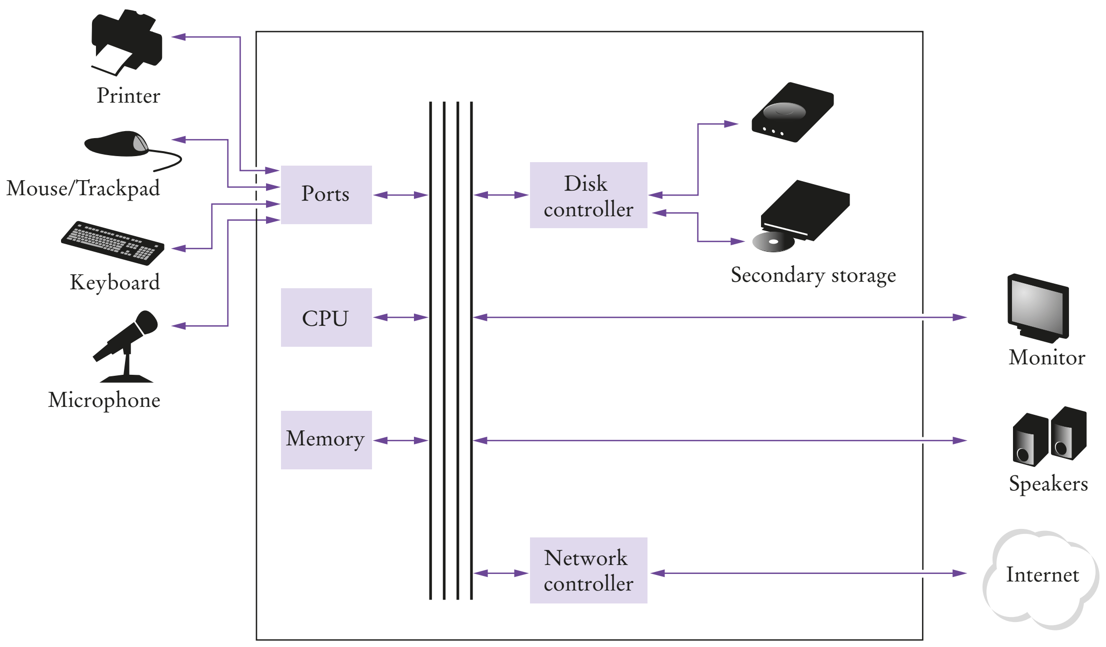


Figure 3 Schematic Design of a Personal Computer

SELF CHECK

4. Where is a program stored when it is not currently running?
5. Which part of the computer carries out arithmetic operations, such as addition and multiplication?
6. A modern smartphone is a computer, comparable to a desktop computer. Which components of a smartphone correspond to those shown in Figure 3?

Practice It Now you can try these exercises at the end of the chapter: R1.2, R1.3.

Computing & Society 1.1 Computers Are Everywhere

When computers were first invented in the 1940s, a computer filled an entire room. The photo below shows the ENIAC (*e*lectronic *n*umerical *i*ntegrator *a*nd *c*omputer), completed in 1946 at the University of Pennsylvania. The ENIAC was used by the military to compute the trajectories of projectiles. Nowadays, computing facilities of search engines, Internet shops, and social networks fill huge buildings called data centers. At the other end of the spectrum, computers are all around us. Your cell phone has a computer inside, as do many credit cards and fare cards for public transit. A modern car has several computers—to control the engine, brakes, lights, and the radio.

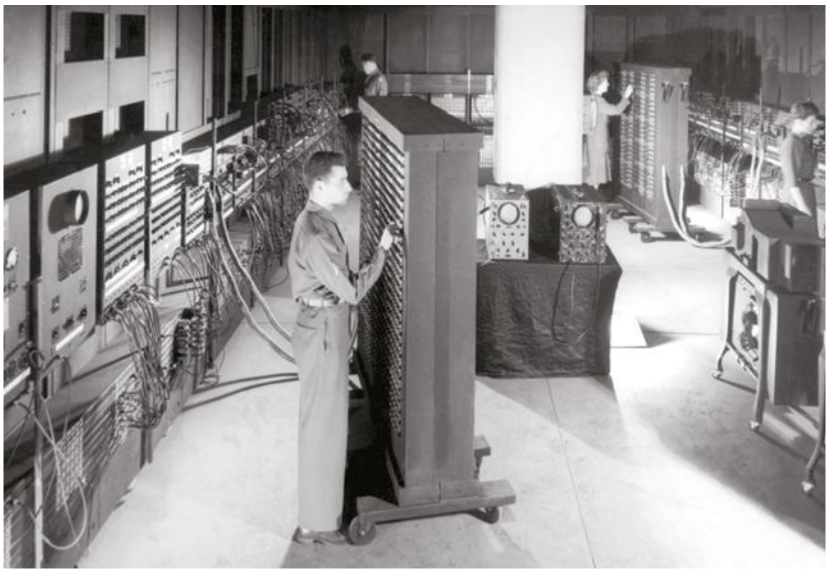
The advent of ubiquitous computing changed many aspects of our lives. Factories used to employ people to do repetitive assembly tasks that are today carried out by computer-controlled robots, operated by a few people who know how to work with those computers. Books, music, and movies nowadays are often consumed on computers, and computers are almost always involved in their production. The book that you are reading right now could not have been written without computers.

Knowing about computers and how to program them has become an essential skill in many careers. Engineers design computer-controlled cars and medical equipment that preserve lives. Computer scientists develop programs that help people come together to support social causes. For example, activists used social networks to share videos showing abuse by repressive regimes, and this information was instrumental in changing public opinion.

As computers, large and small, become ever more embedded in our everyday lives, it is increasingly important for everyone to understand how they work, and how to work with them. As you use this book to learn how to program a computer, you will develop a good understanding of computing fundamentals that will make you a more informed citizen and, perhaps, a computing professional.

© Maurice Savage/Alamy Limited.

This transit card contains a computer.



© UPPA/Photoshot.

The ENIAC

1.3 The Java Programming Language

James Sullivan/Getty Images.

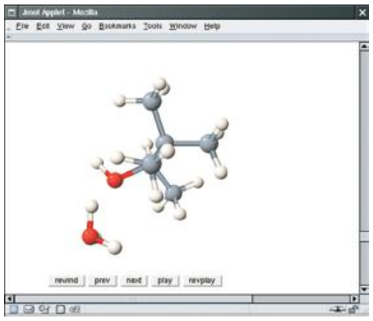
James Gosling

In order to write a computer program, you need to provide a sequence of instructions that the CPU can execute. A computer program consists of a large number of simple CPU instructions, and it is tedious and error-prone to specify them one by one. For that reason, **high-level programming languages** have been created. In a high-level language, you specify the actions that your program should carry out. A **compiler** translates the high-level instructions into the more detailed instructions (called **machine code**)required by the CPU. Many different programming languages have been designed for different purposes.

Java was originally designed for programming consumer devices, but it was first successfully used to write Internet applets.

In 1991, a group led by James Gosling and Patrick Naughton at Sun Microsystems designed a programming language, code-named "Green", for use in consumer devices, such as intelligent television "set-top" boxes. The language was designed to be simple, secure, and usable for many different processor types. No customer was ever found for this technology.

Gosling recounts that in 1994 the team realized, "We could write a really cool browser. It was one of the few things in the client/server mainstream that needed some of the weird things we'd done: architecture neutral, real-time, reliable, secure." Java was introduced to an enthusiastic crowd at the SunWorld exhibition in 1995, together with a browser that ran **applets**—Java code that can be located anywhere on the Internet. The figure at right shows a typical example of an applet.



An Applet for Visualizing Molecules

Since then, Java has grown at a phenomenal rate. Programmers have embraced the language because it is easier to use than its closest rival, C++. In addition, Java has a rich **library** that makes it possible to write portable programs that can bypass proprietary operating systems—a feature that was eagerly sought by those who wanted to be independent of those proprietary systems and was bitterly fought by their vendors. A "micro edition" and an "enterprise edition" of the Java library allow Java programmers to target hardware ranging from smart cards to the largest Internet servers.

Java was designed to be safe and portable, benefiting both Internet users and students.

Because Java was designed for the Internet, it has two attributes that make it very suitable for beginners: safety and portability.

Table 1 Java Versions (since Version 1.0 in 1996)

Version	Year	Important New Features	Version	Year	Important New Features
1.1	1997	Inner classes	5	2004	Generic classes, enhanced `for` loop, auto-boxing, enumerations, annotations
1.2	1998	Swing, Collections framework	6	2006	Library improvements
1.3	2000	Performance enhancements	7	2011	Small language changes and library improvements
1.4	2002	Assertions, XML support	8	2014	Function expressions, streams, new date/time library

Java was designed so that anyone can execute programs in their browser without fear. The safety features of the Java language ensure that a program is terminated if it tries to do something unsafe. Having a safe environment is also helpful for anyone learning Java. When you make an error that results in unsafe behavior, your program is terminated and you receive an accurate error report.

Java programs are distributed as instructions for a virtual machine, making them platform-independent.

The other benefit of Java is portability. The same Java program will run, without change, on Windows, UNIX, Linux, or Macintosh. In order to achieve portability, the Java compiler does not translate Java programs directly into CPU instructions. Instead, compiled Java programs contain instructions for the Java **virtual machine**, a program that simulates a real CPU. Portability is another benefit for the beginning student. You do not have to learn how to write programs for different platforms.

At this time, Java is firmly established as one of the most important languages for general-purpose programming as well as for computer science instruction. However, although Java is a good language for beginners, it is not perfect, for three reasons.

Because Java was not specifically designed for students, no thought was given to making it really simple to write basic programs. A certain amount of technical machinery is necessary to write even the simplest programs. This is not a problem for professional programmers, but it can be a nuisance for beginning students. As you learn how to program in Java, there will be times when you will be asked to be satisfied with a preliminary explanation and wait for more complete detail in a later chapter.

Java has been extended many times during its life—see Table 1. In this book, we assume that you have Java version 7 or later.

Java has a very large library. Focus on learning those parts of the library that you need for your programming projects.

Finally, you cannot hope to learn all of Java in one course. The Java language itself is relatively simple, but Java contains a vast set of *library packages* that are required to write useful programs. There are packages for graphics, user-interface design, cryptography, networking, sound, database storage, and many other purposes. Even expert Java programmers cannot hope to know the contents of all of the packages—they just use those that they need for particular projects.

Using this book, you should expect to learn a good deal about the Java language and about the most important packages. Keep in mind that the central goal of this book is not to make you memorize Java minutiae, but to teach you how to think about programming.

SELF CHECK

7. What are the two most important benefits of the Java language?

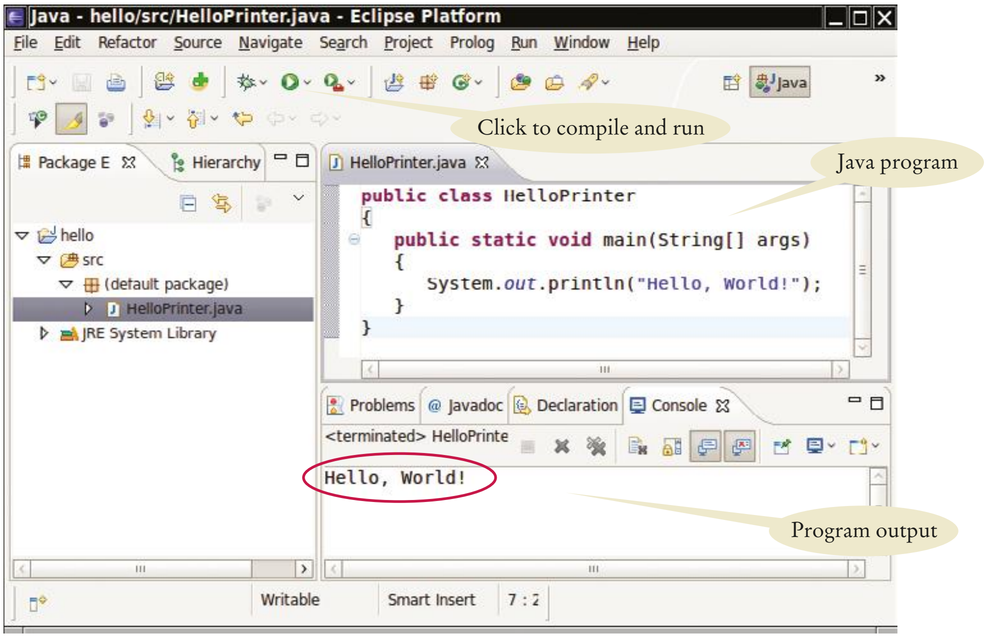
8. How long does it take to learn the entire Java library?

Practice It Now you can try this exercise at the end of the chapter: R1.5.

1.4 Becoming Familiar with Your Programming Environment

Set aside time to become familiar with the programming environment that you will use for your class work.

Many students find that the tools they need as programmers are very different from the software with which they are familiar. You should spend some time making yourself familiar with your programming environment. Because computer systems vary widely, this book can only give an outline of the steps you need to follow. It is a good idea to participate in a hands-on lab, or to ask a knowledgeable friend to give you a tour.



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

```
Hello, World!
```

Figure 4 Running the `HelloPrinter` Program in an Integrated Development Environment

Step 1 Start the Java development environment.

An editor is a program for entering and modifying text, such as a Java program.

Computer systems differ greatly in this regard. On many computers there is an **integrated development environment** in which you can write and test your programs. On other computers you first launch an **editor**, a program that functions like a word processor, in which you can enter your Java instructions; you then open a *console window* and type commands to execute your program. You need to find out how to get started with your environment.

Step 2 Write a simple program.

The traditional choice for the very first program in a new programming language is a program that displays a simple greeting: “Hello, World!”. Let us follow that tradition. Here is the “Hello, World!” program in Java:

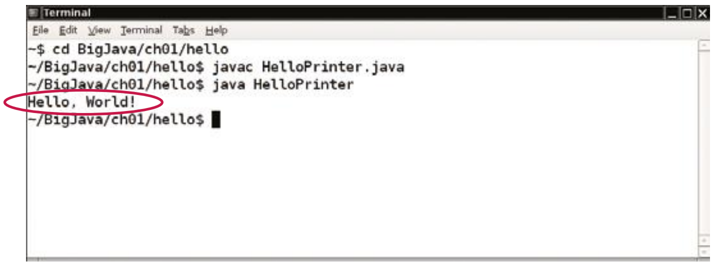
```
public class HelloPrinter
{
   public static void main(String[] args)
   {
      System.out.println("Hello, World!");
   }
}
```

We will examine this program in the next section.

No matter which programming environment you use, you begin your activity by typing the program statements into an editor window.

Create a new file and call it `HelloPrinter.java`, using the steps that are appropriate for your environment. (If your environment requires that you supply a project name in addition to the file name, use the name `hello` for the project.) Enter the program instructions *exactly* as they are given above. Alternatively, locate the electronic copy in this book’s companion code and paste it into your editor.

Figure 5
Running the `HelloPrinter` Program in a Console Window



```
~$ cd BigJava/ch01/hello
~/BigJava/ch01/hello$ javac HelloPrinter.java
~/BigJava/ch01/hello$ java HelloPrinter
Hello, World!
~/BigJava/ch01/hello$
```

Java is case sensitive. You must be careful about distinguishing between upper- and lowercase letters.

As you write this program, pay careful attention to the various symbols, and keep in mind that Java is **case sensitive**. You must enter upper- and lowercase letters exactly as they appear in the program listing. You cannot type `MAIN` or `PrintLn`. If you are not careful, you will run into problems—see Common Error 1.2 on page 15.

Step 3 Run the program.

The process for running a program depends greatly on your programming environment. You may have to click a button or enter some commands. When you run the test program, the message

```
Hello, World!
```

will appear somewhere on the screen (see Figures 4 and 5).

The Java compiler translates source code into class files that contain instructions for the Java virtual machine.

In order to run your program, the Java compiler translates your **source files** (that is, the statements that you wrote) into *class files*. (A class file contains instructions for the Java virtual machine.) After the compiler has translated your **source code** into virtual machine instructions, the virtual machine executes them. During execution, the virtual machine accesses a library of pre-written code, including the implementations of the `System` and `PrintStream` classes that are necessary for displaying the program's output. Figure 6 summarizes the process of creating and running a Java program. In some programming environments, the compiler and virtual machine are essentially invisible to the programmer—they are automatically executed whenever you ask to run a Java program. In other environments, you need to launch the compiler and virtual machine explicitly.

Step 4 Organize your work.

As a programmer, you write programs, try them out, and improve them. You store your programs in **files**. Files are stored in **folders** or **directories**. A folder can contain

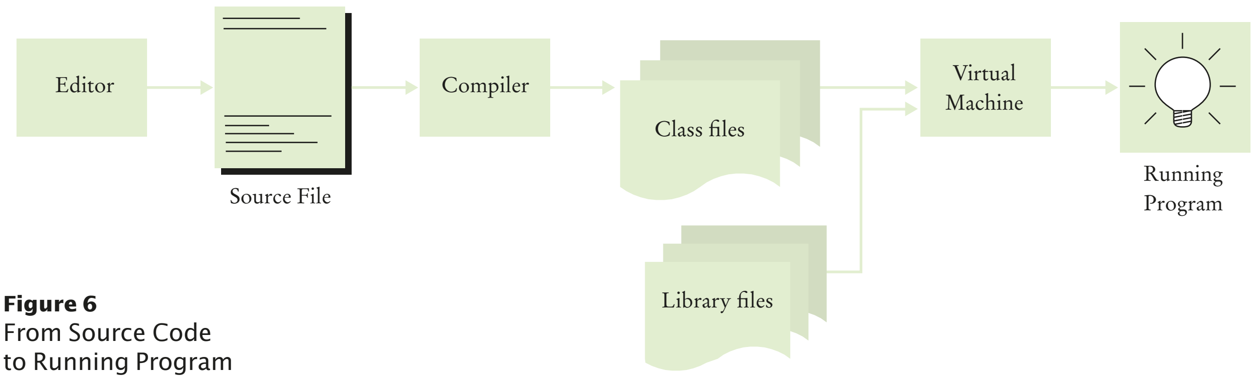
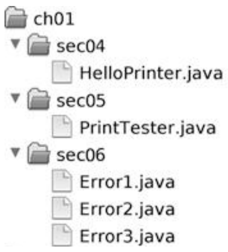


Figure 6
From Source Code to Running Program



ch01
sec04
HelloPrinter.java
sec05
PrintTester.java
sec06
Error1.java
Error2.java
Error3.java

Figure 7
A Folder Hierarchy

files as well as other folders, which themselves can contain more files and folders (see Figure 7). This hierarchy can be quite large, and you need not be concerned with all of its branches. However, you should create folders for organizing your work. It is a good idea to make a separate folder for your programming coursework. Inside that folder, make a separate folder for each program.

Some programming environments place your programs into a default location if you don't specify a folder yourself. In that case, you need to find out where those files are located.

Be sure that you understand where your files are located in the folder hierarchy. This information is essential when you submit files for grading, and for making backup copies (see Programming Tip 1.1).

SELF CHECK

9. Where is the `HelloPrinter.java` file stored on your computer?

10. What do you do to protect yourself from data loss when you work on programming projects?

Practice It Now you can try this exercise at the end of the chapter: R1.6.

Programming Tip 1.1

Backup Copies

© Tatiana Popova/iStockphoto.

You will spend many hours creating and improving Java programs. It is easy to delete a file by accident, and occasionally files are lost because of a computer malfunction. Retyping the contents of lost files is frustrating and time-consuming. It is therefore crucially important that you learn how to safeguard files and get in the habit of doing so *before* disaster strikes. Backing up files on a memory stick is an easy and convenient storage method for many people. Another increasingly popular form of backup is Internet file storage. Here are a few pointers to keep in mind:

Develop a strategy for keeping backup copies of your work before disaster strikes.

- *Back up often.* Backing up a file takes only a few seconds, and you will hate yourself if you have to spend many hours recreating work that you could have saved easily. I recommend that you back up your work once every thirty minutes.
- *Rotate backups.* Use more than one directory for backups, and rotate them. That is, first back up onto the first directory. Then back up onto the second directory. Then use the third, and then go back to the first. That way you always have three recent backups. If your recent changes made matters worse, you can then go back to the older version.
- *Pay attention to the backup direction.* Backing up involves copying files from one place to another. It is important that you do this right—that is, copy from your work location to the backup location. If you do it the wrong way, you will overwrite a newer file with an older version.
- *Check your backups once in a while*. Double-check that your backups are where you think they are. There is nothing more frustrating than to find out that the backups are not there when you need them.
- *Relax, then restore.* When you lose a file and need to restore it from a backup, you are likely to be in an unhappy, nervous state. Take a deep breath and think through the recovery process before you start. It is not uncommon for an agitated computer user to wipe out the last backup when trying to restore a damaged file.

VIDEO EXAMPLE 1.1 **Compiling and Running a Program**

See a demonstration of how to compile and run a simple Java program. Go to `wiley.com/go/bjlo2videos` to view Video Example 1.1.

1.5 Analyzing Your First Program

In this section, we will analyze the first Java program in detail. Here again is the source code:

sec04/HelloPrinter.java

```java
public class HelloPrinter
{
   public static void main(String[] args)
   {
      // Display a greeting in the console window

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

The line

```
public class HelloPrinter
```

indicates the declaration of a **class** called `HelloPrinter`.

Classes are the fundamental building blocks of Java programs.

Every Java program consists of one or more classes. Classes are the fundamental building blocks of Java programs. You will have to wait until Chapter 8 for a full explanation of classes.

The word `public` denotes that the class is usable by the “public”. You will later encounter `private` features.

In Java, every source file can contain at most one public class, and the name of the public class must match the name of the file containing the class. For example, the class `HelloPrinter` must be contained in a file named `HelloPrinter.java`.

The construction

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

Every Java application contains a class with a `main` method. When the application starts, the instructions in the `main` method are executed.

declares a **method** called `main`. A method contains a collection of programming instructions that describe how to carry out a particular task. Every Java application must have a **`main` method**. Most Java programs contain other methods besides `main`, and you will see in Chapter 5 how to write other methods.

Each class contains declarations of methods. Each method contains a sequence of instructions.

The term `static` is explained in more detail in Chapter 8, and the meaning of `String[] args` is covered in Chapter 7. At this time, simply consider

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

as a part of the "plumbing" that is required to create a Java program. Our first program has all instructions inside the `main` method of the class.

The `main` method contains one or more instructions called **statements**. Each statement ends in a semicolon (`;`). When a program runs, the statements in the `main` method are executed one by one.

In our example program, the `main` method has a single statement:

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

This statement prints a line of text, namely "Hello, World!". In this statement, we *call* a method which, for reasons that we will not explain here, is specified by the rather long name `System.out.println`.

We do not have to implement this method—the programmers who wrote the Java library already did that for us. We simply want the method to perform its intended task, namely to print a value.

A method is called by specifying the method and its arguments.

Whenever you call a method in Java, you need to specify

1. The method you want to use (in this case, `System.out.println`).
2. Any values the method needs to carry out its task (in this case, `"Hello, World!"`). The technical term for such a value is an **argument**. Arguments are enclosed in parentheses. Multiple arguments are separated by commas.

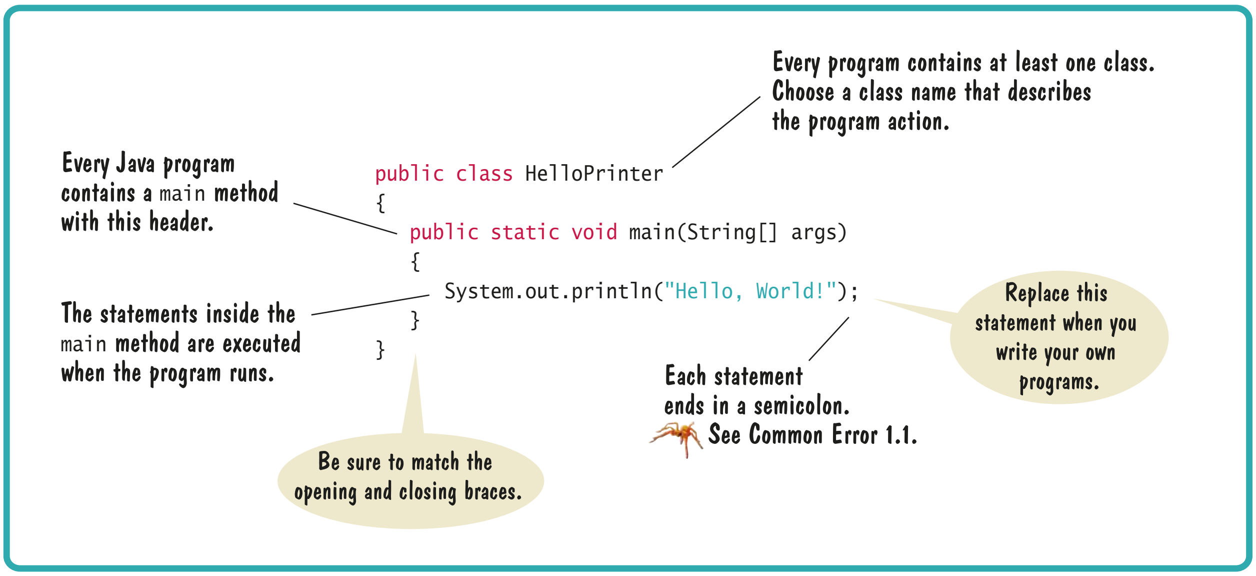
A sequence of characters enclosed in quotation marks

```
"Hello, World!"
```

A string is a sequence of characters enclosed in quotation marks.

is called a **string**. You must enclose the contents of the string inside quotation marks so that the compiler knows you literally mean `"Hello, World!"`. There is a reason for this requirement. Suppose you need to print the word *main*. By enclosing it in quotation marks, `"main"`, the compiler knows you mean the sequence of characters `m a i n`, not the method named `main`. The rule is simply that you must enclose all text strings in quotation marks, so that the compiler considers them plain text and does not try to interpret them as program instructions.

Syntax 1.1 Java Program



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

Every program contains at least one class. Choose a class name that describes the program action.
Every Java program contains a main method with this header.
The statements inside the main method are executed when the program runs.
Replace this statement when you write your own programs.
Each statement ends in a semicolon. See Common Error 1.1.
Be sure to match the opening and closing braces.

You can also print numerical values. For example, the statement

```
System.out.println(3 + 4);
```

evaluates the expression 3 + 4 and displays the number 7.

The `System.out.println` method prints a string or a number and then starts a new line. For example, the sequence of statements

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

prints two lines of text:

```
Hello
World!
```

FULL CODE EXAMPLE
Go to `wiley.com/go/bjlo2code` to download a program to demonstrate print commands.

There is a second method, `System.out.print`, that you can use to print an item without starting a new line. For example, the output of the two statements

```
System.out.print("00");
System.out.println(3 + 4);
```

is the single line

```
007
```

SELF CHECK

11. How do you modify the `HelloPrinter` program to greet you instead?

12. How would you modify the `HelloPrinter` program to print the word "Hello" vertically?

13. Would the program continue to work if you replaced line 7 with this statement?

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

14. What does the following set of statements print?

```
System.out.print("My lucky number is");
System.out.println(3 + 4 + 5);
```

15. What do the following statements print?

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

Practice It Now you can try these exercises at the end of the chapter: R1.7, R1.8, E1.5, E1.8.

Common Error 1.1

Omitting Semicolons

In Java every statement must end in a semicolon. Forgetting to type a semicolon is a common error. It confuses the compiler, because the compiler uses the semicolon to find where one statement ends and the next one starts. The compiler does not use line breaks or closing braces to recognize the end of statements. For example, the compiler considers

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

a single statement, as if you had written

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

Then it doesn't understand that statement, because it does not expect the word `System` following the closing parenthesis after `"Hello"`.

The remedy is simple. Scan every statement for a terminating semicolon, just as you would check that every English sentence ends in a period. However, do not add a semicolon at the end of `public class Hello` or `public static void main`. These lines are not statements.

1.6 Errors

Experiment a little with the `HelloPrinter` program. What happens if you make a typing error such as

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

© Martin Carlsson/iStockphoto.

Programmers spend a fair amount of time fixing compile-time and run-time errors.

A compile-time error is a violation of the programming language rules that is detected by the compiler.

In the first case, the compiler will complain. It will say that it has no clue what you mean by `ou`. The exact wording of the error message is dependent on your development environment, but it might be something like "Cannot find symbol ou". This is a **compile-time error**. Something is wrong according to the rules of the language and the compiler finds it. For this reason, compile-time errors are often called **syntax errors**. When the compiler finds one or more errors, it refuses to translate the program into Java virtual machine instructions, and as a consequence you have no program that you can run. You must fix the error and compile again. In fact, the compiler is quite picky, and it is common to go through several rounds of fixing compile-time errors before compilation succeeds for the first time.

If the compiler finds an error, it will not simply stop and give up. It will try to report as many errors as it can find, so you can fix them all at once.

Sometimes, an error throws the compiler off track. Suppose, for example, you forget the quotation marks around a string: `System.out.println(Hello, World!)`. The compiler will not complain about the missing quotation marks. Instead, it will report "Cannot find symbol Hello". Unfortunately, the compiler is not very smart and it does not realize that you meant to use a string. It is up to you to realize that you need to enclose strings in quotation marks.

The error in the second line above is of a different kind. The program will compile and run, but its output will be wrong. It will print

```
Hello, Word!
```

A run-time error causes a program to take an action that the programmer did not intend.

This is a **run-time error**. The program is syntactically correct and does something, but it doesn't do what it is supposed to do. Because run-time errors are caused by logical flaws in the program, they are often called **logic errors**.

This particular run-time error did not include an error message. It simply produced the wrong output. Some kinds of run-time errors are so severe that they generate an **exception**: an error message from the Java virtual machine. For example, if your program includes the statement

```
System.out.println(1 / 0);
```

you will get a run-time error message "Division by zero".

FULL CODE EXAMPLE

Go to `wiley.com/go/bjlo2code` to download three programs that illustrate errors.

During program development, errors are unavoidable. Once a program is longer than a few lines, it would require superhuman concentration to enter it correctly without slipping up once. You will find yourself omitting semicolons or quotation marks more often than you would like, but the compiler will track down these problems for you.

Run-time errors are more troublesome. The compiler will not find them—in fact, the compiler will cheerfully translate any program as long as its syntax is correct—

but the resulting program will do something wrong. It is the responsibility of the program author to test the program and find any run-time errors.

SELF CHECK

16. Suppose you omit the `""` characters around `Hello, World!` from the `HelloPrinter.java` program. Is this a compile-time error or a run-time error?
17. Suppose you change `println` to `printline` in the `HelloPrinter.java` program. Is this a compile-time error or a run-time error?
18. Suppose you change `main` to `hello` in the `HelloPrinter.java` program. Is this a compile-time error or a run-time error?
19. When you used your computer, you may have experienced a program that "crashed" (quit spontaneously) or "hung" (failed to respond to your input). Is that behavior a compile-time error or a run-time error?
20. Why can't you test a program for run-time errors when it has compiler errors?

Practice It Now you can try these exercises at the end of the chapter: R1.9, R1.10, R1.11.

Common Error 1.2

Misspelling Words

If you accidentally misspell a word, then strange things may happen, and it may not always be completely obvious from the error messages what went wrong. Here is a good example of how simple spelling errors can cause trouble:

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

This class declares a method called `Main`. The compiler will not consider this to be the same as the `main` method, because `Main` starts with an uppercase letter and the Java language is case sensitive. Upper- and lowercase letters are considered to be completely different from each other, and to the compiler `Main` is no better match for `main` than `rain`. The compiler will cheerfully compile your `Main` method, but when the Java virtual machine reads the compiled file, it will complain about the missing `main` method and refuse to run the program. Of course, the message "missing main method" should give you a clue where to look for the error.

If you get an error message that seems to indicate that the compiler or virtual machine is on the wrong track, check for spelling and capitalization. If you misspell the name of a symbol (for example, `ou` instead of `out`), the compiler will produce a message such as "cannot find symbol ou". That error message is usually a good clue that you made a spelling error.

1.7 Problem Solving: Algorithm Design

You will soon learn how to program calculations and decision making in Java. But before we look at the mechanics of implementing computations in the next chapter, let's consider how you can describe the steps that are necessary for finding the solution to a problem.

1.7.1 The Algorithm Concept

You may have run across advertisements that encourage you to pay for a computerized service that matches you up with a love partner. Think how this might work. You fill out a form and send it in. Others do the same. The data are processed by a computer program. Is it reasonable to assume that the computer can perform the task of finding the best match for you? Suppose your younger brother, not the computer, had all the forms on his desk. What instructions could you give him? You can't say, "Find the best-looking person who likes inline skating and browsing the Internet". There is no objective standard for good looks, and your brother's opinion (or that of a computer program analyzing the photos of prospective partners) will likely be different from yours. If you can't give written instructions for someone to solve the problem, there is no way the computer can magically find the right solution. The computer can only do what you tell it to do. It just does it faster, without getting bored or exhausted.

© mammamaart/iStockphoto.

Finding the perfect partner is not a problem that a computer can solve.

For that reason, a computerized match-making service cannot guarantee to find the optimal match for you. Instead, you may be presented with a set of potential partners who share common interests with you. That is a task that a computer program can solve.

In order for a computer program to provide an answer to a problem that computes an answer, it must follow a sequence of steps that is

- Unambiguous
- Executable
- Terminating

An algorithm for solving a problem is a sequence of steps that is unambiguous, executable, and terminating.

The step sequence is *unambiguous* when there are precise instructions for what to do at each step and where to go next. There is no room for guesswork or personal opinion. A step is *executable* when it can be carried out in practice. For example, a computer can list all people that share your hobbies, but it can't predict who will be your life-long partner. Finally, a sequence of steps is *terminating* if it will eventually come to an end. A program that keeps working without delivering an answer is clearly not useful.

A sequence of steps that is unambiguous, executable, and terminating is called an **algorithm**. Although there is no algorithm for finding a partner, many problems do have algorithms for solving them. The next section gives an example.

© Claudiad/iStockphoto.

An algorithm is a recipe for finding a solution.

1.7.2 An Algorithm for Solving an Investment Problem

Consider the following investment problem:

> You put $10,000 into a bank account that earns 5 percent interest per year. How many years does it take for the account balance to be double the original?

Could you solve this problem by hand? Sure, you could. You figure out the balance as follows:

year	interest	balance
0		10000
1	10000.00 x 0.05 = 500.00	10000.00 + 500.00 = 10500.00
2	10500.00 x 0.05 = 525.00	10500.00 + 525.00 = 11025.00
3	11025.00 x 0.05 = 551.25	11025.00 + 551.25 = 11576.25
4	11576.25 x 0.05 = 578.81	11576.25 + 578.81 = 12155.06

You keep going until the balance is at least $20,000. Then the last number in the year column is the answer.

Of course, carrying out this computation is intensely boring to you or your younger brother. But computers are very good at carrying out repetitive calculations quickly and flawlessly. What is important to the computer is a description of the steps for finding the solution. Each step must be clear and unambiguous, requiring no guesswork. Here is such a description:

```
Set year to 0, balance to 10000.
```

year	interest	balance
0		10000

```
While the balance is less than $20,000
    Add 1 to the year.
    Set the interest to balance x 0.05 (i.e., 5 percent interest).
    Add the interest to the balance.
```

year	interest	balance
0		10000
1	500.00	10500.00
14	942.82	19799.32
15	989.96	20789.28

```
Report year as the answer.
```

These steps are not yet in a language that a computer can understand, but you will soon learn how to formulate them in Java. This informal description is called **pseudocode**. We examine the rules for writing pseudocode in the next section.

1.7.3 Pseudocode

Pseudocode is an informal description of a sequence of steps for solving a problem.

There are no strict requirements for pseudocode because it is read by human readers, not a computer program. Here are the kinds of pseudocode statements and how we will use them in this book:

- Use statements such as the following to describe how a value is set or changed:

```
total cost = purchase price + operating cost
Multiply the balance value by 1.05.
Remove the first and last character from the word.
```

- Describe decisions and repetitions as follows:

```
If total cost 1 < total cost 2
While the balance is less than $20,000
For each picture in the sequence
```

  Use indentation to indicate which statements should be selected or repeated:

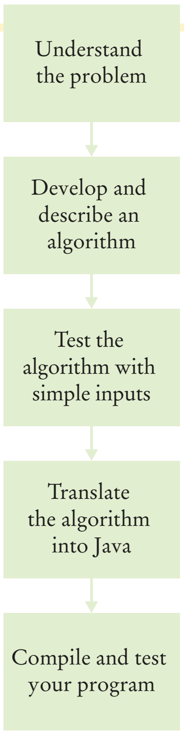
```
For each car
    operating cost = 10 x annual fuel cost
    total cost = purchase price + operating cost
```

  Here, the indentation indicates that both statements should be executed for each car.

- Indicate results with statements such as:

```
Choose car1.
Report year as the answer.
```

1.7.4 From Algorithms to Programs


Understand the problem
Develop and describe an algorithm
Test the algorithm with simple inputs
Translate the algorithm into Java
Compile and test your program

In Section 1.7.2, we developed pseudocode for finding how long it takes to double an investment. Let's double-check that the pseudocode represents an algorithm; that is, that it is unambiguous, executable, and terminating.

Our pseudocode is unambiguous. It simply tells how to update values in each step. The pseudocode is executable because we use a fixed interest rate. Had we said to use the actual interest rate that will be charged in years to come, and not a fixed rate of 5 percent per year, the instructions would not have been executable. There is no way for anyone to know what the interest rate will be in the future. It requires a bit of thought to see that the steps are terminating: With every step, the balance goes up by at least $500, so eventually it must reach $20,000.

Therefore, we have found an algorithm to solve our investment problem, and we know we can find the solution by programming a computer. The existence of an algorithm is an essential prerequisite for programming a task. You need to first discover and describe an algorithm for the task before you start programming (see Figure 8). In the chapters that follow, you will learn how to express algorithms in the Java language.

Figure 8 The Software Development Process

SELF CHECK

21. Suppose the interest rate was 20 percent. How long would it take for the investment to double?

22. Suppose your cell phone carrier charges you $29.95 for up to 300 minutes of calls, and $0.45 for each additional minute, plus 12.5 percent taxes and fees. Give an algorithm to compute the monthly charge from a given number of minutes.

23. Consider the following pseudocode for finding the most attractive photo from a sequence of photos:

```
Set "the best so far" to the first photo.
For each photo in the sequence
    If the photo is more attractive than "the best so far"
        Set "the best so far" to the photo.
Report "the best so far" as the most attractive photo in the sequence.
```

Is this an algorithm that will find the most attractive photo?

24. Suppose each photo in Self Check 23 had a price tag. Give an algorithm for finding the most expensive photo.

25. Suppose you have a random sequence of black and white marbles and want to rearrange it so that the black and white marbles are grouped together. Consider this algorithm:

```
Repeat until sorted
    Locate the first black marble that is preceded by a white marble, and switch them.
```

What does the algorithm do with the sequence ○●○●●? Spell out the steps until the algorithm stops.

26. Suppose you have a random sequence of colored marbles. Consider this pseudocode:

```
Repeat until sorted
    Locate the first marble that is preceded by a marble of a different color, and switch them.
```

Why is this not an algorithm?

Practice It Now you can try these exercises at the end of the chapter: R1.16, E1.4, P1.1.

HOW TO 1.1 Describing an Algorithm with Pseudocode

This is the first of many "How To" sections in this book that give you step-by-step procedures for carrying out important tasks in developing computer programs.

Before you are ready to write a program in Java, you need to develop an algorithm—a method for arriving at a solution for a particular problem. Describe the algorithm in pseudocode—a sequence of precise steps formulated in English. To illustrate, we'll devise an algorithm for this problem:

Problem Statement You have the choice of buying one of two cars. One is more fuel efficient than the other, but also more expensive. You know the price and fuel efficiency (in miles per gallon, mpg) of both cars. You plan to keep the car for ten years. Assume a price of $4 per gallon of gas and usage of 15,000 miles per year. You will pay cash for the car and not worry about financing costs. Which car is the better deal?

© dlewis33/iStockphoto.

Step 1 Determine the inputs and outputs.

In our sample problem, we have these inputs:

- **purchase price1** and **fuel efficiency1**, the price and fuel efficiency (in mpg) of the first car
- **purchase price2** and **fuel efficiency2**, the price and fuel efficiency of the second car

We simply want to know which car is the better buy. That is the desired output.

Step 2 Break down the problem into smaller tasks.

For each car, we need to know the total cost of driving it. Let's do this computation separately for each car. Once we have the total cost for each car, we can decide which car is the better deal.

The total cost for each car is **purchase price + operating cost**.

We assume a constant usage and gas price for ten years, so the operating cost depends on the cost of driving the car for one year.

The operating cost is **10 x annual fuel cost**.

The annual fuel cost is **price per gallon x annual fuel consumed**.

The annual fuel consumed is **annual miles driven / fuel efficiency**. For example, if you drive the car for 15,000 miles and the fuel efficiency is 15 miles/gallon, the car consumes 1,000 gallons.

Step 3 Describe each subtask in pseudocode.

In your description, arrange the steps so that any intermediate values are computed before they are needed in other computations. For example, list the step

```
total cost = purchase price + operating cost
```

after you have computed **operating cost**.

Here is the algorithm for deciding which car to buy:

```
For each car, compute the total cost as follows:
    annual fuel consumed = annual miles driven / fuel efficiency
    annual fuel cost = price per gallon x annual fuel consumed
    operating cost = 10 x annual fuel cost
    total cost = purchase price + operating cost
If total cost of car1 < total cost of car2
    Choose car1.
Else
    Choose car2.
```

Step 4 Test your pseudocode by working a problem.

We will use these sample values:

Car 1: $25,000, 50 miles/gallon
Car 2: $20,000, 30 miles/gallon

Here is the calculation for the cost of the first car:

```
annual fuel consumed = annual miles driven / fuel efficiency = 15000 / 50 = 300
annual fuel cost = price per gallon x annual fuel consumed = 4 x 300 = 1200
operating cost = 10 x annual fuel cost = 10 x 1200 = 12000
total cost = purchase price + operating cost = 25000 + 12000 = 37000
```

Similarly, the total cost for the second car is $40,000. Therefore, the output of the algorithm is to choose car 1.

The following Worked Example demonstrates how to use the concepts in this chapter and the steps in the How To to solve another problem. In this case, you will see how to develop an algorithm for laying tile in an alternating pattern of colors. You should read the Worked Example to review what you have learned, and for help in tackling another problem.

In future chapters, Worked Examples are provided for you on the book's companion Web site. A brief description of the problem tackled in the example will appear with a reminder to download it from `www.wiley.com/go/bjlo2examples`. You will find any code related to the Worked Example included with the book's companion code for the chapter. When you see the Worked Example description, download the example and the code to learn how the problem was solved.

WORKED EXAMPLE 1.1 Writing an Algorithm for Tiling a Floor

Problem Statement Write an algorithm for tiling a rectangular bathroom floor with alternating black and white tiles measuring 4 × 4 inches. The floor dimensions, measured in inches, are multiples of 4.

Step 1 Determine the inputs and outputs.

The inputs are the floor dimensions (length × width), measured in inches. The output is a tiled floor.

© rban/iStockphoto.

Step 2 Break down the problem into smaller tasks.

A natural subtask is to lay one row of tiles. If you can solve that task, then you can solve the problem by laying one row next to the other, starting from a wall, until you reach the opposite wall.

How do you lay a row? Start with a tile at one wall. If it is white, put a black one next to it. If it is black, put a white one next to it. Keep going until you reach the opposite wall. The row will contain **width / 4** tiles.

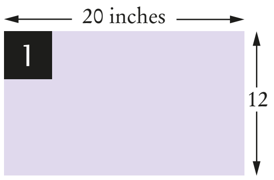
Step 3 Describe each subtask in pseudocode.

In the pseudocode, you want to be more precise about exactly where the tiles are placed.

```
Place a black tile in the northwest corner.
While the floor is not yet filled
    Repeat width / 4 – 1 times
        Place a tile east of the previously placed tile. If the previously placed tile was white, pick a black one;
            otherwise, a white one.
    Locate the tile at the beginning of the row that you just placed. If there is space to the south, place a tile of
    the opposite color below it.
```

Step 4 Test your pseudocode by working a problem.

Suppose you want to tile an area measuring 20 × 12 inches. The first step is to place a black tile in the northwest corner.


20 inches
1
12