

2

Introduction to Java Applications

Objectives

- To be able to write simple Java applications.
- To be able to use input and output statements.
- To become familiar with primitive data types.
- To understand basic memory concepts.
- To be able to use arithmetic operators.
- To understand the precedence of arithmetic operators.
- To be able to write decision-making statements.
- To be able to use relational and equality operators.

Comment is free, but facts are sacred.

C. P. Scott

The creditor hath a better memory than the debtor.

James Howell

When faced with a decision, I always ask, "What would be the most fun?"

Peggy Walker

*He has left his body to science—
and science is contesting the will.*

David Frost

Classes struggle, some classes triumph, others are eliminated.

Mao Zedong

Equality, in a social sense, may be divided into that of condition and that of rights.

James Fenimore Cooper



Outline

- 2.1 Introduction
- 2.2 A First Program in Java: Printing a Line of Text
 - 2.2.1 Compiling and Executing your First Java Application
- 2.3 Modifying Our First Java Program
 - 2.3.1 Displaying a Single Line of Text with Multiple Statements
 - 2.3.2 Displaying Multiple Lines of Text with a Single Statement
- 2.4 Displaying Text in a Dialog Box
- 2.5 Another Java Application: Adding Integers
- 2.6 Memory Concepts
- 2.7 Arithmetic
- 2.8 Decision Making: Equality and Relational Operators
- 2.9 (Optional Case Study) Thinking About Objects: Examining the Problem Statement

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

2.1 Introduction

The Java language facilitates a disciplined approach to computer program design. We now introduce Java programming and present examples that illustrate several important features of Java. Each example is analyzed one line at a time. In this chapter and Chapter 3, we present two program types in Java—*applications* and *applets*. In Chapter 4 and Chapter 5, we present a detailed treatment of *program development* and *program control* in Java.

2.2 A First Program in Java: Printing a Line of Text

Java uses notations that may appear strange to nonprogrammers. We begin by considering a simple *application* that displays a line of text. An application is a program that executes using the **java** interpreter (discussed later in this section). The program and its output are shown in Fig. 2.1.

This program illustrates several important features of the Java language. We consider each line of the program in detail. Each program we present in this book has line numbers included for the reader's convenience; line numbers are not part of actual Java programs. Line 9 does the “real work” of the program, namely displaying the phrase **Welcome to Java Programming!** on the screen. But let us consider each line in order. Line 1,

```
// Fig. 2.1: Welcome1.java
```

begins with `//`, indicating that the remainder of the line is a *comment*. Programmers insert comments to *document* programs and improve program readability. Comments also help other people read and understand a program. Comments do not cause the computer to perform any action when the program is run. The Java compiler ignores comments. We begin every program with a comment indicating the figure number and file name (line 1).

```
1 // Fig. 2.1: Welcome1.java
2 // A first program in Java.
3
4 public class Welcome1 {
5
6     // main method begins execution of Java application
7     public static void main( String args[] )
8     {
9         System.out.println( "Welcome to Java Programming!" );
10    } // end method main
11
12 } // end class Welcome1
```

Welcome to Java Programming!

Fig. 2.1 A first program in Java.



Good Programming Practice 2.1

Use comments to clarify difficult concepts used in a program.

A comment that begins with `//` is called a *single-line comment*, because the comment terminates at the end of the current line. A `//` comment can also begin in the middle of a line and continue until the end of that line.

Multiple-line comments can be written in two other forms. For example,

```
/* This is a multiple
   line comment. It can be
   split over many lines */
```

is a comment that can spread over several lines. This type of comment begins with delimiter `/*` and ends with delimiter `*/`; this type of comment may be called a *multiple-line comment*. All text between the delimiters of the comment is ignored by the compiler. A similar form of comment called a *documentation comment* is delimited by `/**` and `*/`.



Common Programming Error 2.1

Forgetting one of the delimiters of a multiple-line comment is a syntax error.

Java absorbed comments delimited with `/*` and `*/` from the C programming language and single-line comments delimited with `//` from the C++ programming language. Java programmers generally use C++-style single-line comments in preference to C-style comments. Throughout this book, we use C++-style single-line comments. The documentation comment syntax (`/**` and `*/`) is special to Java. It enables programmers to embed documentation for their programs directly in the programs. The **javadoc** utility program (provided by Sun Microsystems with the Java 2 Software Development Kit) reads those comments from the program and uses them to prepare your program's documentation. There are subtle issues to using **javadoc**-style comments properly. We do not use **javadoc**-style comments in the programs presented in this book. However, **javadoc**-style comments are explained thoroughly in Appendix F.

Line 2,

```
// A first program in Java.
```

is a single-line comment that describes the purpose of the program.



Good Programming Practice 2.2

Every program should begin with a comment describing the purpose of the program.

Line 3 is simply a blank line. Programmers use blank lines and space characters to make programs easier to read. Together, blank lines, space characters and tab characters are known as *white space*. (Space characters and tabs are known specifically as *white-space characters*.) Such characters are ignored by the compiler. We discuss conventions for using white-space characters in this chapter and the next several chapters, as these spacing conventions are needed in many Java programs.



Good Programming Practice 2.3

Use blank lines, space characters and tab characters to enhance program readability.

Line 4,

```
public class Welcome1 {
```

begins a *class definition* for class **Welcome1**. Every program in Java consists of at least one class definition that is defined by you—the programmer. These classes are known as *programmer-defined classes*, or *user-defined classes*. The **class** keyword introduces a class definition in Java and is immediately followed by the *class name* (**Welcome1** in this program). Keywords (or *reserved words*) are reserved for use by Java (we discuss the various keywords throughout the text) and are always spelled with all lowercase letters. The complete list of Java keywords is shown in Fig. 4.2.

By convention, all class names in Java begin with a capital letter and have a capital letter for every word in the class name (e.g., **SampleClassName**). The name of the class is called an *identifier*. An identifier is a series of characters consisting of letters, digits, underscores (`_`) and dollar signs (`$`) that does not begin with a digit and does not contain spaces. Some valid identifiers are **Welcome1**, **\$value**, **_value**, **m_inputField1** and **button7**. The name **7button** is not a valid identifier, because it begins with a digit, and the name **input field** is not a valid identifier, because it contains a space. Java is *case sensitive*—i.e., uppercase and lowercase letters are different, so **a1** and **A1** are different identifiers.



Common Programming Error 2.2

Java is case sensitive. Not using the proper uppercase and lowercase letters for an identifier is normally a syntax error.



Good Programming Practice 2.4

By convention, you should always begin a class name with a capital letter.



Good Programming Practice 2.5

When reading a Java program, look for identifiers that start with capital letters. These identifiers normally represent Java classes.



Software Engineering Observation 2.1

Avoid using identifiers that contain dollar signs (\$), as the compiler often uses dollar signs to create identifier names.

In Chapter 2 through Chapter 7, every class we define begins with the **public** keyword. For now, we will simply require this keyword. The **public** keyword is discussed in detail in Chapter 8. Also in that chapter, we discuss classes that do not begin with keyword **public**. [Note: Several times early in this text, we ask you to mimic certain Java features we introduce as you write your own Java programs. We specifically do this when it is not yet important for you to know all of the details of a feature in order for you to use that feature in Java. All programmers initially learn how to program by mimicking what other programmers have done before them. For each detail we ask you to mimic, we indicate where the full discussion will be presented later in the text.]

When you save your **public** class definition in a file, the file name must be the class name followed by the “.java” file-name extension. For our application, the file name is **Welcome1.java**. All Java class definitions are stored in files ending with the file-name extension “.java.”



Common Programming Error 2.3

It is an error for a **public** class if the file name is not identical to the class name (plus the .java extension) in terms of both spelling and capitalization. Therefore, it is also an error for a file to contain two or more **public** classes.



Common Programming Error 2.4

It is an error not to end a file name with the .java extension for a file containing an application's class definition. If the extension is missing, the Java compiler will not be able to compile the class definition.

A left brace (at the end of line 4), {, begins the body of every class definition. A corresponding right brace (in line 13 in this program), }, must end each class definition. Notice that lines 6–11 are indented. This indentation is one of the spacing conventions mentioned earlier. We define each spacing convention as a *Good Programming Practice*.



Good Programming Practice 2.6

Whenever you type an opening left brace, {, in your program, immediately type the closing right brace, }, then reposition the cursor between the braces to begin typing the body. This practice helps prevent errors due to missing braces.



Good Programming Practice 2.7

Indent the entire body of each class definition one “level” of indentation between the left brace, {, and the right brace, }, that define the body of the class. This format emphasizes the structure of the class definition and helps make the class definition easier to read.



Good Programming Practice 2.8

Set a convention for the indent size you prefer, and then uniformly apply that convention. The Tab key may be used to create indents, but tab stops may vary between editors. We recommend using three spaces to form a level of indent.



Common Programming Error 2.5

If braces do not occur in matching pairs, the compiler indicates an error.

Line 5 is a blank line, inserted for program readability. Line 6,

```
// main method begins execution of Java application
```

is a single-line comment indicating the purpose of lines 6–11 of the program.

Line 7,

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

is a part of every Java application. Java applications begin executing at **main**. The parentheses after **main** indicate that **main** is a program building block called a *method*. Java class definitions normally contain one or more methods. For a Java application class, exactly one of those methods must be called **main** and must be defined as shown on line 7; otherwise, the **java** interpreter will not execute the application. Methods are able to perform tasks and return information when they complete their tasks. The **void** keyword indicates that this method will perform a task (displaying a line of text, in this program), but will not return any information when it completes its task. Later, we will see that many methods return information when they complete their task. Methods are explained in detail in Chapter 6. For now, simply mimic **main**'s first line in your Java applications.

The left brace, **{**, on line 8 begins the *body of the method definition*. A corresponding right brace, **}**, must end the method definition's body (line 11 of the program). Notice that the line in the body of the method is indented between the braces.



Good Programming Practice 2.9

*Indent the entire body of each method definition one “level” of indentation between the left brace, **{**, and the right brace, **}**, that define the body of the method. This format makes the structure of the method stand out and helps make the method definition easier to read.*

Line 9,

```
System.out.println( "Welcome to Java Programming!" );
```

instructs the computer to perform an *action*, namely to print the *string* of characters contained between the double quotation marks. A string is sometimes called a *character string*, a *message* or a *string literal*. We refer to characters between double quotation marks generically as *strings*. White-space characters in strings are not ignored by the compiler.

System.out is known as the *standard output object*. **System.out** allows Java applications to display strings and other types of information in the *command window* from which the Java application executes. In Microsoft Windows 95/98/ME, the command window is the *MS-DOS prompt*. In Microsoft Windows NT/2000, the command window is the *Command Prompt (cmd.exe)*. In UNIX, the command window is normally called a *command window*, a *command tool*, a *shell tool* or a *shell*. On computers running an operating system that does not have a command window (such as a Macintosh), the **java** interpreter normally displays a window containing the information the program displays.

Method **System.out.println** displays (or prints) a line of text in the command window. When **System.out.println** completes its task, it automatically positions the *output cursor* (the location where the next character will be displayed) to the beginning of the next line in the command window. (This move of the cursor is similar to you pressing the *Enter* key when typing in a text editor—the cursor appears at the beginning of the next line in your file.)

The entire line, including **System.out.println**, its *argument* in the parentheses (the string) and the *semicolon* (**;**), is a *statement*. Every statement must end with a semicolon (also known as the *statement terminator*). When the statement on line 9 of our program executes, it displays the message **Welcome to Java Programming!** in the command window.



Common Programming Error 2.6

Omitting the semicolon at the end of a statement is a syntax error. A syntax error occurs when the compiler cannot recognize a statement. The compiler normally issues an error message to help the programmer identify and fix the incorrect statement. Syntax errors are violations of the language rules. Syntax errors are also called compile errors, compile-time errors or compilation errors, because the compiler detects them during the compilation phase. You will be unable to execute your program until you correct all of the syntax errors in it.



Testing and Debugging Tip 2.1

When the compiler reports a syntax error, the error may not be on the line number indicated by the error message. First, check the line for which the error was reported. If that line does not contain syntax errors, check the preceding several lines in the program.

Some programmers find it difficult when reading and/or writing a program to match the left and right braces (**{** and **}**) that delimit the body of a class definition or a method definition. For this reason, some programmers prefer to include a single-line comment after a closing right brace (**}**) that ends a method definition and after a closing right brace that ends a class definition. For example, line 11,

```
} // end method main
```

specifies the closing right brace (**}**) of method **main**, and line 13,

```
} // end class Welcome1
```

specifies the closing right brace (**}**) of class **Welcome1**. Each comment indicates the method or class that the right brace terminates. We use such comments through Chapter 6 to help beginning programmers determine where each program component terminates. After Chapter 6, we use such comments when pairs of braces contain many statements, which makes the closing braces difficult to identify.



Good Programming Practice 2.10

Some programmers prefer to follow the closing right brace (**}**) of a method body or class definition with a single-line comment indicating the method or class definition to which the brace belongs. This comment improves program readability.

2.2.1 Compiling and Executing your First Java Application

We are now ready to compile and execute our program. To compile the program, we open a command window, change to the directory where the program is stored and type

```
javac Welcome1.java
```

If the program contains no syntax errors, the preceding command creates a new file called **Welcome1.class** containing the Java bytecodes that represent our application. These bytecodes will be interpreted by the **java** interpreter when we tell it to execute the program, as shown in the Microsoft Windows 2000 **Command Prompt** of Fig. 2.2.

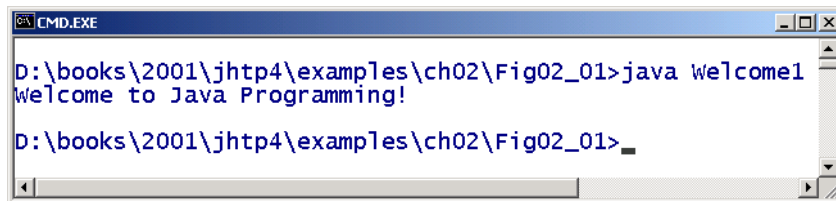


Fig. 2.2 Executing **Welcome1** in a Microsoft Windows 2000 **Command Prompt**.

In the command prompt of Figure 2.2, we typed

```
java Welcome1
```

to launch the **java** interpreter and indicate that it should load the “**.class**” file for class **Welcome1**. Note that the “**.class**” file-name extension is omitted from the preceding command; otherwise the interpreter will not execute the program. The interpreter automatically calls method **main**. Next, the statement on line 7 of **main** displays “**Welcome to Java Programming!**”



Testing and Debugging Tip 2.2

The Java compiler generates syntax error messages when the syntax of a program is incorrect. When you are learning how to program, sometimes it is helpful to “break” a working program so you can see the error messages produced by the compiler. Then, when you encounter that error message again, you will have an idea of the error’s cause. Try removing a semicolon or curly brace from the program of Fig. 2.1, then recompile the program to see the error messages generated by the omission.

2.3 Modifying Our First Java Program

This section continues our introduction to Java programming with two examples that modify the example in Fig. 2.1 to print text on one line by using multiple statements and to print text on several lines by using a single statement.

2.3.1 Displaying a Single Line of Text with Multiple Statements

Welcome to Java Programming! can be displayed using several methods. Class **Welcome2**, shown in Fig. 2.3, uses two statements to produce the same output as that shown in Fig. 2.1.

Most of the program is identical to that of Fig. 2.1, so we discuss only the changes here. Line 2,

```
// Printing a line of text with multiple statements.
```

is a single-line comment stating the purpose of this program. Line 4 begins the definition of class **Welcome2**.

Lines 9–10 of method **main**,

```
System.out.print( "Welcome to " );
System.out.println( "Java Programming!" );
```

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01

```
1 // Fig. 2.3: Welcome2.java
2 // Printing a line of text with multiple statements.
3
4 public class Welcome2 {
5
6     // main method begins execution of Java application
7     public static void main( String args[] )
8     {
9         System.out.print( "Welcome to " );
10        System.out.println( "Java Programming!" );
11
12    } // end method main
13
14 } // end class Welcome2
```

```
Welcome to Java Programming
```

Fig. 2.3 Printing a line of text with multiple statements.

display one line of text in the command window. The first statement uses **System.out**'s method **print** to display a string. The difference between **print** and **println** is that, after displaying its argument, **print** does not position the output cursor at the beginning of the next line in the command window; the next character the program displays in the command window will appear immediately after the last character that **print** displays. Thus, line 10 positions the first character in its argument, "J," immediately after the last character that line 9 displays (the space character at the end of the string on line 9). Each **print** or **println** statement resumes displaying characters from where the last **print** or **println** statement stopped displaying characters.

2.3.2 Displaying Multiple Lines of Text with a Single Statement

A single statement can display multiple lines by using *newline characters*. Newline characters are "special characters" that indicate to **System.out**'s **print** and **println** methods when they should position the output cursor to the beginning of the next line in the command window. Figure 2.4 outputs four lines of text, using newline characters to determine when to begin each new line.

Most of the program is identical to those of Fig. 2.1 and Fig. 2.3, so we discuss only the changes here. Line 2,

```
// Printing multiple lines of text with a single statement.
```

is a single-line comment stating the purpose of this program. Line 4 begins the definition of class **Welcome3**.

Line 9,

```
System.out.println( "Welcome\nto\nJava\nProgramming!" );
```

displays four separate lines of text in the command window. Normally, the characters in a string are displayed exactly as they appear in the double quotes. Notice, however, that the

two characters `\` and `n` are not printed on the screen. The *backslash* (`\`) is called an *escape character*. It indicates that a “special character” is to be output. When a backslash appears in a string of characters, Java combines the next character with the backslash to form an *escape sequence*. The escape sequence `\n` is the *newline character*. When a newline character appears in a string being output with **System.out**, the newline character causes the screen’s output cursor to move to the beginning of the next line in the command window. Some other common escape sequences are listed in Fig. 2.5.

```

1  // Fig. 2.4: Welcome3.java
2  // Printing multiple lines of text with a single statement.
3
4  public class Welcome3 {
5
6      // main method begins execution of Java application
7      public static void main( String args[] )
8      {
9          System.out.println( "Welcome\n to\n Java\n Programming!" );
10
11      } // end method main
12
13 } // end class Welcome3

```

```

Welcome
to
Java
Programming!

```

Fig. 2.4 Printing multiple lines of text with a single statement.

Escape sequence	Description
<code>\n</code>	Newline. Position the screen cursor to the beginning of the next line.
<code>\t</code>	Horizontal tab. Move the screen cursor to the next tab stop.
<code>\r</code>	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line. Any characters output after the carriage return overwrite the characters previously output on that line.
<code>\\</code>	Backslash. Used to print a backslash character.
<code>\"</code>	Double quote. Used to print a double-quote character. For example,
	<pre>System.out.println("\"in quotes\"");</pre> displays <pre>"in quotes"</pre>

Fig. 2.5 Some common escape sequences.

2.4 Displaying Text in a Dialog Box

Although the first several programs presented in this chapter display output in the command window, many Java applications use windows or *dialog boxes* (also called *dialogs*) to display output. For example, World Wide Web browsers such as Netscape Navigator or Microsoft Internet Explorer display Web pages in their own windows. Email programs typically allow you to type and read messages in a window provided by the email program. Typically, dialog boxes are windows in which programs display important messages to the user of the program. Java's class **JOptionPane** provides prepackaged dialog boxes that enable programs to display messages to users. Figure 2.6 displays the same string as in Fig. 2.4 in a predefined dialog box known as a *message dialog*.

One of the great strengths of Java is its rich set of predefined classes that programmers can reuse rather than “reinventing the wheel.” We use many of these classes throughout the book. Java's numerous predefined classes are grouped into categories of related classes called *packages*. The packages are referred to collectively as the *Java class library*, or the *Java applications programming interface (Java API)*. The packages of the Java API are split into *core packages* and *extension packages*. The names of the packages begin with either “**java**” (core packages) or “**javax**” (extension packages). Many of the core and extension packages are included as part of the Java 2 Software Development Kit. We overview these included packages in Chapter 6. As Java continues to evolve, new packages are developed as extension packages. These extensions often can be downloaded from **java.sun.com** and used to enhance Java's capabilities. In this example, we use class **JOptionPane**, which Java defines for us in package **javax.swing**.

Line 4,

```
// Java extension packages
```

is a single-line comment indicating the section of the program in which we specify **import** statements for classes in Java's extension packages. In every program that specifies **import** statements, we separate the import statements into the following groups: Java core packages (for package names starting with **java**), Java extension packages (for package names starting with **javax**) and Deitel packages (for our own packages defined later in the book).

Line 5,

```
import javax.swing.JOptionPane; // import class JOptionPane
```

is an **import** statement. The compiler uses **import** statements to identify and load classes used in a Java program. When you use classes from the Java API, the compiler attempts to ensure that you use them correctly. The **import** statements help the compiler locate the classes you intend to use. For each new class we use from the Java API, we indicate the package in which you can find that class. This package information is important. It helps you locate descriptions of each package and class in the *Java API documentation*. A Web-based version of this documentation can be found at

java.sun.com/j2se/1.3/docs/api/index.html

Also, you can download this documentation to your own computer from

java.sun.com/j2se/1.3/docs.html

```

1  // Fig. 2.6: Welcome4.java
2  // Printing multiple lines in a dialog box
3
4  // Java extension packages
5  import javax.swing.JOptionPane; // import class JOptionPane
6
7  public class Welcome4 {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         JOptionPane.showMessageDialog(
13             null, "Welcome\n\tto\n\tJava\n\tProgramming!" );
14
15         System.exit( 0 ); // terminate application
16
17     } // end method main
18
19 } // end class Welcome4

```

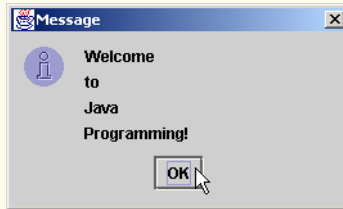


Fig. 2.6 Displaying multiple lines in a dialog box.

We will provide an overview of the use of this documentation with the downloads and resources for *Java How to Program, Fourth Edition* on our Web site, www.deitel.com. Packages are discussed in detail in Chapter 8, Object-Based Programming.



Common Programming Error 2.7

All **import** statements must appear before the class definition. Placing an **import** statement inside a class definition's body or after a class definition is a syntax error.

Line 5 tells the compiler to load the **JOptionPane** class from the **javax.swing** package. This package contains many classes that help Java programmers define *graphical user interfaces (GUIs)* for their application. *GUI components* facilitate data entry by the user of your program and formatting or presentation of data outputs to the user of your program. For example, Fig. 2.7 contains a Netscape Navigator window. In the window, there is a bar containing *menus* (**File**, **Edit**, **View**, etc.), called a *menu bar*. Below the menu bar is a set of *buttons* that each have a defined task in Netscape Navigator. Below the buttons there is a *text field* in which the user can type the name of the World Wide Web site to visit. The menus, buttons and text fields are part of Netscape Navigator's GUI. They enable you to interact with the Navigator program. Java contains classes that implement the GUI components described here and others that will be described in Chapter 12, Basic Graphical User Interface Components, and Chapter 13, Advanced Graphical User Interface Components.

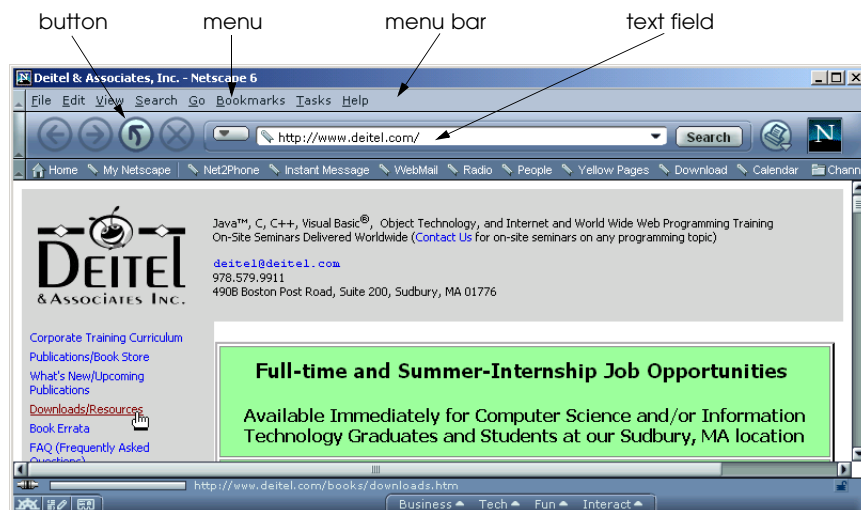


Fig. 2.7 A sample Netscape Navigator window with GUI components.

In method **main** of Fig. 2.6, lines 12–13,

```
JOptionPane.showMessageDialog(
    null, "Welcome\nto\nJava\nProgramming!" );
```

indicate a call to method **showMessageDialog** of class **JOptionPane**. The method requires two arguments. When a method requires multiple arguments, the arguments are separated with *commas* (,). Until we discuss **JOptionPane** in detail in Chapter 13, the first argument will always be the keyword **null**. The second argument is the string to display. The first argument helps the Java application determine where to position the dialog box. When the first argument is **null**, the dialog box appears in the center of the computer screen. Most applications you use on your computer execute in their own window (e.g., email programs, Web browsers and word processors). When such an application displays a dialog box, it normally appears in the center of the application window, which is not necessarily the center of the screen. Later in this book, you will see more elaborate applications in which the first argument to method **showMessageDialog** will cause the dialog box to appear in the center of the application window, rather than the center of the screen.



Good Programming Practice 2.11

Place a space after each comma (,) in an argument list, to make programs more readable.

Method **JOptionPane.showMessageDialog** is a special method of class **JOptionPane** called a *static method*. Such methods are always called by using their class name followed by a dot operator (.) and the method name, as in

```
ClassName.methodName( arguments )
```

Many of the predefined methods we introduce early in this book are **static** methods. We ask you to mimic this syntax for calling **static** methods until we discuss them in detail in Chapter 8, Object-Based Programming.

Executing the statement at lines 12–13 displays the dialog box in Fig. 2.8. The *title bar* of the dialog contains the string **Message**, to indicate that the dialog is presenting a message to the user. The dialog box automatically includes an **OK** button that allows the user to *dismiss (hide) the dialog* by pressing the button. This is accomplished by positioning the *mouse cursor* (also called the *mouse pointer*) over the **OK** button and clicking the left mouse button.

Remember that all statements in Java end with a semicolon (;). Therefore, lines 12–13 represent one statement. Java allows large statements to be split over many lines. However, you cannot split a statement in the middle of an identifier or in the middle of a string.



Common Programming Error 2.8

Splitting a statement in the middle of an identifier or a string is a syntax error.

Line 15,

```
System.exit( 0 ); // terminate application
```

uses **static** method **exit** of class **System** to terminate the application. In any application that displays a graphical user interface, this line is required in order to terminate the application. Notice once again the syntax used to call the method—the class name (**System**), a dot (.) and the method name (**exit**). Remember that identifiers starting with capital letters normally represent class names. So, you can assume that **System** is a class. Class **System** is part of the package **java.lang**. Notice that class **System** is not imported with an **import** statement at the beginning of the program. By default, package **java.lang** is imported in every Java program. Package **java.lang** is the only package in the Java API for which you are not required to provide an **import** statement.

The argument **0** to method **exit** indicates that the application has terminated successfully. (A nonzero value normally indicates that an error has occurred.) This value is passed to the command window that executed the program. The argument is useful if the program is executed from a *batch file* (on Windows 95/98/ME/NT/2000 systems) or a *shell script* (on UNIX/Linux systems). Batch files and shell scripts often execute several programs in sequence. When the first program ends, the next program begins execution automatically. It is possible to use the argument to method **exit** in a batch file or shell script to determine whether other programs should execute. For more information on batch files or shell scripts, see your operating system's documentation.

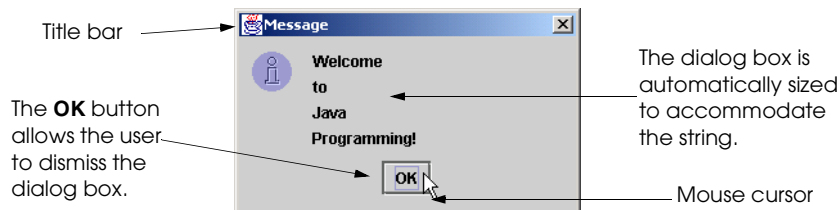


Fig. 2.8 Message dialog box.



Common Programming Error 2.9

Forgetting to call **System.exit** in an application that displays a graphical user interface prevents the program from terminating properly. This omission normally results in the command window preventing you from typing any other commands. Chapter 14 discusses in more detail the reason that **System.exit** is required in GUI-based applications.

2.5 Another Java Application: Adding Integers

Our next application inputs two *integers* (whole numbers, like -22, 7 and 1024) typed by a user at the keyboard, computes the sum of the values and displays the result. This program uses another predefined dialog box from class **JOptionPane** called an *input dialog* that allows the user to input a value for use in the program. The program also uses a message dialog to display the sum of the integers. Figure 2.9 shows the application and sample screen captures.

```

1  // Fig. 2.9: Addition.java
2  // An addition program.
3
4  // Java extension packages
5  import javax.swing.JOptionPane; // import class JOptionPane
6
7  public class Addition {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         String firstNumber; // first string entered by user
13         String secondNumber; // second string entered by user
14         int number1; // first number to add
15         int number2; // second number to add
16         int sum; // sum of number1 and number2
17
18         // read in first number from user as a String
19         firstNumber =
20             JOptionPane.showInputDialog( "Enter first integer" );
21
22         // read in second number from user as a String
23         secondNumber =
24             JOptionPane.showInputDialog( "Enter second integer" );
25
26         // convert numbers from type String to type int
27         number1 = Integer.parseInt( firstNumber );
28         number2 = Integer.parseInt( secondNumber );
29
30         // add the numbers
31         sum = number1 + number2;
32
33         // display the results
34         JOptionPane.showMessageDialog(
35             null, "The sum is " + sum, "Results",
36             JOptionPane.PLAIN_MESSAGE );

```

Fig. 2.9 An addition program “in action” (part 1 of 2).

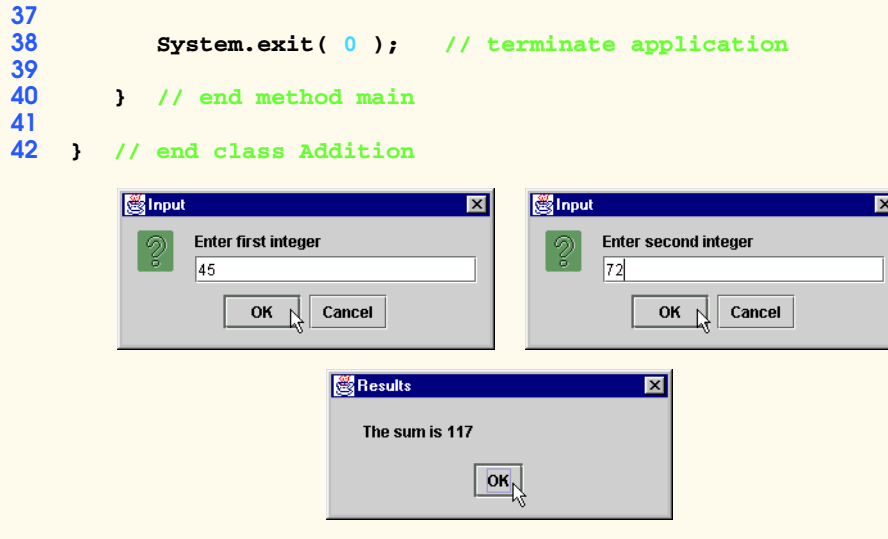


Fig. 2.9 An addition program “in action” (part 2 of 2).

Lines 1 and 2,

```

// Fig. 2.9: Addition.java
// An addition program.

```

are single-line comments stating the figure number, file name and purpose of the program.

Line 4,

```

// Java extension packages

```

is a single-line comment specifying that the next line imports a class from the Java extension packages.

Line 5,

```

import javax.swing.JOptionPane;    // import class JOptionPane

```

indicates that the compiler should load class **JOptionPane** for use in this application.

As stated earlier, every Java program consists of at least one class definition. Line 7,

```

public class Addition {

```

begins the definition of class **Addition**. The file name for this **public** class must be **Addition.java**.

Remember that all class definitions start with an opening left brace (at the end of line 7), **{**, and end with a closing right brace, **}** (in line 42).

As stated earlier, every application begins execution with method **main** (lines 10–40). The left brace (line 11) marks the beginning of **main**’s body and the corresponding right brace (line 36) marks the end of **main**’s body.

Lines 12–13,

```

String firstNumber;    // first string entered by user
String secondNumber;   // second string entered by user

```

are *declarations*. The words **firstNumber** and **secondNumber** are the names of *variables*. A variable is a location in the computer's memory where a value can be stored for use by a program. All variables must be declared with a name and a data type before they can be used in a program. This declaration specifies that the variables **firstNumber** and **secondNumber** are data of type **String** (located in package **java.lang**), which means that the variables will hold strings. A variable name can be any valid identifier. Like statements, declarations end with a semicolon (;). Notice the single-line comments at the end of each line. This use and placement of the comments is a common practice used by programmers to indicate the purpose of each variable in the program.



Good Programming Practice 2.12

Choosing meaningful variable names helps a program to be self-documenting (i.e., it becomes easier to understand the program simply by reading it rather than by reading manuals or viewing an excessive number of comments).



Good Programming Practice 2.13

By convention, variable-name identifiers begin with a lowercase letter. As with class names, every word in the name after the first word should begin with a capital letter. For example, identifier **firstNumber** has a capital **N** in its second word, **Number**.



Good Programming Practice 2.14

Some programmers prefer to declare each variable on a separate line. This format allows for easy insertion of a descriptive comment next to each declaration.



Software Engineering Observation 2.2

Java automatically imports classes from package **java.lang**, such as class **String**. Therefore, **import** statements are not required for classes in package **java.lang**.

Declarations can be split over several lines, with each variable in the declaration separated by a comma (i.e., a *comma-separated list* of variable names). Several variables of the same type may be declared in one declaration or in multiple declarations. Lines 12–13 can also be written as follows:

```
String firstNumber, // first string entered by user
      secondNumber; // second string entered by user
```

Lines 14–16,

```
int number1; // first number to add
int number2; // second number to add
int sum;     // sum of number1 and number2
```

declare that variables **number1**, **number2** and **sum** are data of type **int**, which means that these variables will hold *integer* values (whole numbers such as 7, -11, 0 and 31,914). We will soon discuss the data types **float** and **double**, for specifying real numbers (numbers with decimal points, such as 3.4, 0.0 and -11.19), and variables of type **char**, for specifying character data. A **char** variable may hold only a single lowercase letter, a single uppercase letter, a single digit or a single special character (such as **x**, **\$**, **7** and *****) and escape sequences (such as the newline character, **\n**). Java is capable of representing characters from many other spoken languages. Types such as **int**, **double** and **char** are often called *primitive data types*, or *built-in data types*. Primitive-type names are keywords;

thus, they must appear in all lowercase letters. Chapter 4 summarizes the eight primitive types (**boolean**, **char**, **byte**, **short**, **int**, **long**, **float** and **double**).

Line 18 is a single-line comment indicating that the next statement reads the first number from the user. Lines 19–20,

```
firstNumber =  
    JOptionPane.showInputDialog( "Enter first integer" );
```

reads from the user a **String** representing the first of the two integers to add. Method **JOptionPane.showInputDialog** displays the input dialog in Fig. 2.10.

The argument to **showInputDialog** indicates what the user should type in the text field. This message is called a *prompt*, because it directs the user to take a specific action. The user types characters in the text field, and then clicks the **OK** button or presses the *Enter* key to return the string to the program. (If you type and nothing appears in the text field, position the mouse pointer in the text field and click the left mouse button to activate the text field.) Unfortunately, Java does not provide a simple form of input that is analogous to displaying output in the command window with **System.out**'s method **print** and **println**. For this reason, we normally receive input from a user through a GUI component (an input dialog box in this program).

Technically, the user can type anything in the text field of the input. Our program assumes that the user follows directions and enters a valid integer value. In this program, if the user either types a noninteger value or clicks the **Cancel** button in the input dialog, a runtime logic error will occur. Chapter 14, Exception Handling, discusses how to make your programs more robust by enabling them to handle such errors. This is also known as making your program *fault tolerant*.

The result of **JOptionPane** method **showInputDialog** (a **String** containing the characters typed by the user) is given to variable **firstNumber** by using the *assignment operator*, **=**. The statement (lines 19–20) is read as “**firstNumber** gets the value of **JOptionPane.showInputDialog("Enter first integer")**.” The **=** operator is called a *binary operator*, because it has two *operands*: **firstNumber** and the result of the expression **JOptionPane.showInputDialog("Enter first integer")**. This whole statement is called an *assignment statement*, because it assigns a value to a variable. The expression to the right side of the assignment operator, **=**, is always evaluated first. In this case, the program calls method **showInputDialog**, and the value input by the user is assigned to **firstNumber**.

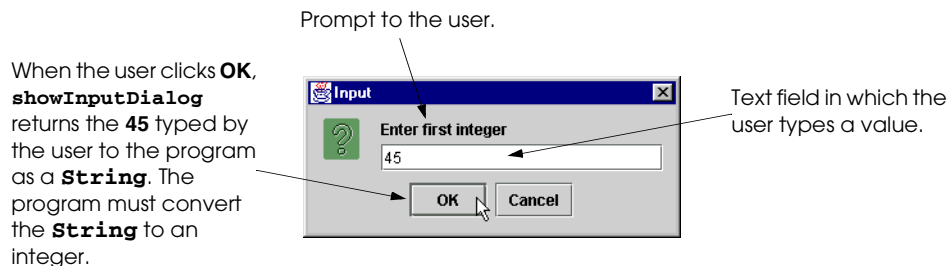


Fig. 2.10 Input dialog box.

Line 22 is a single-line comment indicating that the next statement reads the second number from the user.

Lines 23–24,

```
secondNumber =
    JOptionPane.showInputDialog( "Enter second integer" );
```

display an input dialog in which the user types a **String** representing the second of the two integers to add.

Lines 27–28,

```
number1 = Integer.parseInt( firstNumber );
number2 = Integer.parseInt( secondNumber );
```

convert the two **Strings** input by the user to **int** values that the program can use in a calculation. Method **Integer.parseInt** (a **static** method of class **Integer**) converts its **String** argument to an integer. Class **Integer** is defined in package **java.lang**. Line 27 assigns the **int** (integer) value that **Integer.parseInt** returns to variable **number1**. Line 28 assigns the **int** (integer) value that **Integer.parseInt** returns to variable **number2**.

Line 31,

```
sum = number1 + number2;
```

is an assignment statement that calculates the sum of the variables **number1** and **number2** and assigns the result to variable **sum** by using the assignment operator, **=**. The statement is read as, “**sum** gets the value of **number1 + number2**.” Most calculations are performed in assignment statements. When the program encounters the addition operation, it uses the values stored in the variables **number1** and **number2** to perform the calculation. In the preceding statement, the addition operator is a binary operator: its two operands are **number1** and **number2**.



Good Programming Practice 2.15

Place spaces on either side of a binary operator. This format makes the operator stand out and makes the program more readable.

After the calculation has been performed, lines 34–36,

```
JOptionPane.showMessageDialog(
    null, "The sum is " + sum, "Results",
    JOptionPane.PLAIN_MESSAGE );
```

use method **JOptionPane.showMessageDialog** to display the result of the addition. This new version of **JOptionPane** method **showMessageDialog** requires four arguments. As in Fig. 2.6, the **null** first argument indicates that the message dialog will appear in the center of the screen. The second argument is the message to display. In this case, the second argument is the expression

```
"The sum is " + sum
```

which uses the operator `+` to “add” a **String** (the literal `"The sum is "`) and the value of variable `sum` (the **int** variable containing the result of the addition on line 31). Java has a version of the `+` operator for *string concatenation* that concatenates a **String** and a value of another data type (including another **String**); the result of this operation is a new (and normally longer) **String**. If we assume that `sum` contains the integer value `117`, the expression evaluates as follows:

1. Java determines that the two operands of the `+` operator (the string `"The sum is "` and the integer `sum`) are of different types and one of them is a **String**.
2. Java converts `sum` to a **String**.
3. Java appends the **String** representation of `sum` to the end of `"The sum is "`, resulting in the **String** `"The sum is 117"`.

Method `showMessageDialog` displays the resulting **String** in the dialog box. Note that the automatic conversion of integer `sum` occurs only because the addition operation concatenates the **String** literal `"The sum is "` and `sum`. Also, note that the space between `is` and `117` is part of the string `"The sum is "`. String concatenation is discussed in detail in Chapter 10, “Strings and Characters.”



Common Programming Error 2.10

Confusing the `+` operator used for string concatenation with the `+` operator used for addition can lead to strange results. For example, assuming that integer variable `y` has the value 5, the expression `"y + 2 = " + y + 2` results in the string `"y + 2 = 52"`, not `"y + 2 = 7"`, because first the value of `y` is concatenated with the string `"y + 2 = "`, and then the value 2 is concatenated with the new larger string `"y + 2 = 5"`. The expression `"y + 2 = " + (y + 2)` produces the desired result.

The third and fourth arguments of method `showMessageDialog` in Fig. 2.9 represent the string that should appear in the dialog box’s *title bar* and the *dialog box type*, respectively. The fourth argument—`JOptionPane.PLAIN_MESSAGE`—is a value indicating the type of message dialog to display. This type of message dialog does not display an icon to the left of the message. Figure 2.11 illustrates the second and third arguments and shows that there is no icon in the window.

The message dialog types are shown in Fig. 2.12. All message dialog types except `PLAIN_MESSAGE` dialogs display an icon to the user indicating the type of message.

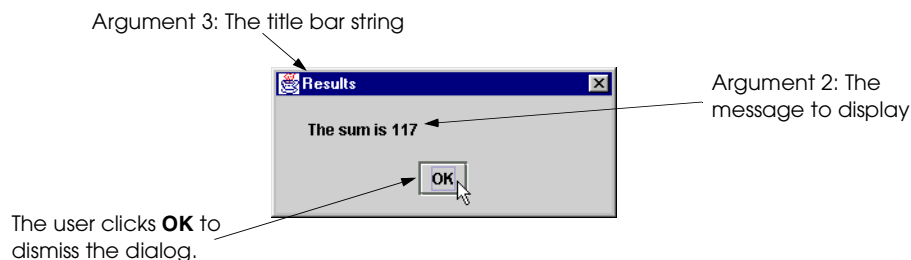


Fig. 2.11 Message dialog box customized with the four-argument version of method `showMessageDialog`.

Message dialog type	Icon	Description
<code>JOptionPane.ERROR_MESSAGE</code>		Displays a dialog that indicates an error to the user.
<code>JOptionPane.INFORMATION_MESSAGE</code>		Displays a dialog with an informational message to the user. The user can simply dismiss the dialog.
<code>JOptionPane.WARNING_MESSAGE</code>		Displays a dialog that warns the user of a potential problem.
<code>JOptionPane.QUESTION_MESSAGE</code>		Displays a dialog that poses a question to the user. This dialog normally requires a response, such as clicking on a Yes or a No button.
<code>JOptionPane.PLAIN_MESSAGE</code>	no icon	Displays a dialog that simply contains a message, with no icon.

Fig. 2.12 `JOptionPane` constants for message dialogs.

2.6 Memory Concepts

Variable names such as **number1**, **number2** and **sum** actually correspond to *locations* in the computer's memory. Every variable has a *name*, a *type*, a *size* and a *value*.

In the addition program of Fig. 2.9, when the statement

```
number1 = Integer.parseInt( firstNumber );
```

executes, the string previously typed by the user in the input dialog and stored in **firstNumber** is converted to an **int** and placed into a memory location to which the name **number1** has been assigned by the compiler. Suppose that the user enters the string **45** as the value for **firstNumber**. The program converts **firstNumber** to an **int**, and the computer places that integer value, **45**, into location **number1**, as shown in Fig. 2.13.

Whenever a value is placed in a memory location, the value replaces the previous value in that location. The previous value is destroyed (i.e., lost).

When the statement

```
number2 = Integer.parseInt( secondNumber );
```

executes, suppose that the user enters the string **72** as the value for **secondNumber**. The program converts **secondNumber** to an **int**, and the computer places that integer value, **72**, into location **number2**. The memory appears as shown in Fig. 2.14.

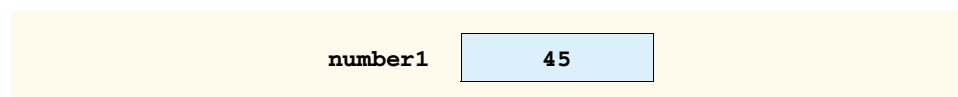


Fig. 2.13 Memory location showing the name and value of variable **number1**.

number1	45
number2	72

Fig. 2.14 Memory locations after storing values for **number1** and **number2**.

After the program of Fig. 2.9 obtains values for **number1** and **number2**, it adds the values and places the sum into variable **sum**. The statement

```
sum = number1 + number2;
```

performs the addition and also replaces **sum**'s previous value. After **sum** has been calculated, memory appears as shown in Fig. 2.15. Note that the values of **number1** and **number2** appear exactly as they did before they were used in the calculation of **sum**. These values were used, but not destroyed, as the computer performed the calculation. Thus, when a value is read from a memory location, the process is nondestructive.

2.7 Arithmetic

Most programs perform arithmetic calculations. The *arithmetic operators* are summarized in Fig. 2.16. Note the use of various special symbols not used in algebra. The *asterisk* (*) indicates multiplication, and the *percent sign* (%) is the *modulus operator*, which is discussed shortly. The arithmetic operators in Fig. 2.16 are binary operators, because they each operate on two operands. For example, the expression **sum + value** contains the binary operator **+** and the two operands **sum** and **value**.

Integer division yields an integer quotient; for example, the expression **7 / 4** evaluates to **1**, and the expression **17 / 5** evaluates to **3**. Note that any fractional part in integer division is simply discarded (i.e., truncated)—no rounding occurs. Java provides the modulus operator, **%**, that yields the remainder after integer division. The expression **x % y** yields the remainder after **x** is divided by **y**. Thus, **7 % 4** yields **3**, and **17 % 5** yields **2**. This operator is most commonly used with integer operands, but also can be used with other arithmetic types. In later chapters, we consider many interesting applications of the modulus operator, such as determining if one number is a multiple of another. There is no arithmetic operator for exponentiation in Java. Chapter 5 shows how to perform exponentiation in Java. [Note: The modulus operator can be used with both integer and floating-point numbers.]

number1	45
number2	72
sum	117

Fig. 2.15 Memory locations after calculating the **sum** of **number1** and **number2**.

Java operation	Arithmetic operator	Algebraic expression	Java expression
Addition	+	$f + 7$	f + 7
Subtraction	-	$p - c$	p - c
Multiplication	*	bm	b * m
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	x / y
Modulus	%	$r \bmod s$	r % s

Fig. 2.16 Arithmetic operators.

Arithmetic expressions in Java must be written in *straight-line form* to facilitate entering programs into the computer. Thus, expressions such as “**a** divided by **b**” must be written as **a / b**, so that all constants, variables and operators appear in a straight line. The following algebraic notation is generally not acceptable to compilers:

$$\frac{a}{b}$$

Parentheses are used in Java expressions in the same manner as in algebraic expressions. For example, to multiply **a** times the quantity **b + c**, we write

$$\mathbf{a * (b + c)}$$

Java applies the operators in arithmetic expressions in a precise sequence determined by the following *rules of operator precedence*, which are generally the same as those followed in algebra:

1. Operators in expressions contained within pairs of parentheses are evaluated first. Thus, *parentheses may be used to force the order of evaluation to occur in any sequence desired by the programmer*. Parentheses are at the highest level of precedence. In cases of *nested* or *embedded* parentheses, the operators in the innermost pair of parentheses are applied first.
2. Multiplication, division and modulus operations are applied next. If an expression contains several multiplication, division or modulus operations, the operators are applied from left to right. Multiplication, division and modulus operators have the same level of precedence.
3. Addition and subtraction operations are applied last. If an expression contains several addition and subtraction operations, the operators are applied from left to right. Addition and subtraction operators have the same level of precedence.

The rules of operator precedence enable Java to apply operators in the correct order. When we say that operators are applied from left to right, we are referring to the *associativity* of the operators. We will see that some operators associate from right to left. Figure 2.17 summarizes these rules of operator precedence. This table will be expanded as additional Java operators are introduced. A complete precedence chart is included in Appendix C.

Operator(s)	Operation(s)	Order of evaluation (precedence)
()	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses on the same level (i.e., not nested), they are evaluated left to right.
*, / and %	Multiplication Division Modulus	Evaluated second. If there are several of this type of operator, they are evaluated from left to right.
+ or -	Addition Subtraction	Evaluated last. If there are several of this type of operator, they are evaluated from left to right.

Fig. 2.17 Precedence of arithmetic operators.

Now, let us consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its Java equivalent.

The following is an example of an arithmetic mean (average) of five terms:

Algebra: $m = \frac{a + b + c + d + e}{5}$

Java: **`m = ( a + b + c + d + e ) / 5;`**

The parentheses are required, because division has higher precedence than that of addition. The entire quantity **`( a + b + c + d + e )`** is to be divided by **`5`**. If the parentheses are erroneously omitted, we obtain **`a + b + c + d + e / 5`**, which evaluates as

$$a + b + c + d + \frac{e}{5}$$

The following is an example of the equation of a straight line:

Algebra: $y = mx + b$

Java: **`y = m * x + b;`**

No parentheses are required. The multiplication operator is applied first, because multiplication has a higher precedence than that of addition. The assignment occurs last, because it has a lower precedence than that of multiplication and division.

The following example contains modulus (%), multiplication, division, addition and subtraction operations:

Algebra: $z = pr\%q + w/x - y$

Java: **`z = p * r % q + w / x - y;`**

6
1
2
4
3
5

The circled numbers under the statement indicate the order in which Java applies the operators. The multiplication, modulus and division operations are evaluated first in left-to-right order (i.e., they associate from left to right), because they have higher precedence than that of addition and subtraction. The addition and subtraction operations are evaluated next. These operations are also applied from left to right.

Not all expressions with several pairs of parentheses contain nested parentheses. For example, the expression

$$a * (b + c) + c * (d + e)$$

does not contain nested parentheses. Rather, these parentheses are on the same level.

To develop a better understanding of the rules of operator precedence, consider the evaluation of a second-degree polynomial ($y = ax^2 + bx + c$):

$$y = a * x * x + b * x + c;$$

6
1
2
4
3
5

The circled numbers under the preceding statement indicate the order in which Java applies the operators. There is no arithmetic operator for exponentiation in Java; x^2 is represented as $x * x$.

Suppose that **a**, **b**, **c** and **x** are initialized as follows: **a** = 2, **b** = 3, **c** = 7 and **x** = 5. Figure 2.18 illustrates the order in which the operators are applied in the preceding second-degree polynomial.

As in algebra, it is acceptable to place unnecessary parentheses in an expression to make the expression clearer. Such unnecessary parentheses are also called *redundant parentheses*. For example, the preceding assignment statement might be parenthesized as follows:

$$y = (a * x * x) + (b * x) + c;$$


Good Programming Practice 2.16

Using parentheses for complex arithmetic expressions, even when the parentheses are not necessary, can make the arithmetic expressions easier to read.

2.8 Decision Making: Equality and Relational Operators

This section introduces a simple version of Java's **if** structure that allows a program to make a decision based on the truth or falsity of some *condition*. If the condition is met (i.e., the condition is *true*), the statement in the body of the **if** structure is executed. If the condition is not met (i.e., the condition is *false*), the body statement does not execute. We will see an example shortly.

Conditions in **if** structures can be formed by using the *equality operators* and *relational operators* summarized in Fig. 2.19. The relational operators all have the same level of precedence and associate from left to right. The equality operators both have the same level of precedence, which is lower than the precedence of the relational operators. The equality operators also associate from left to right.

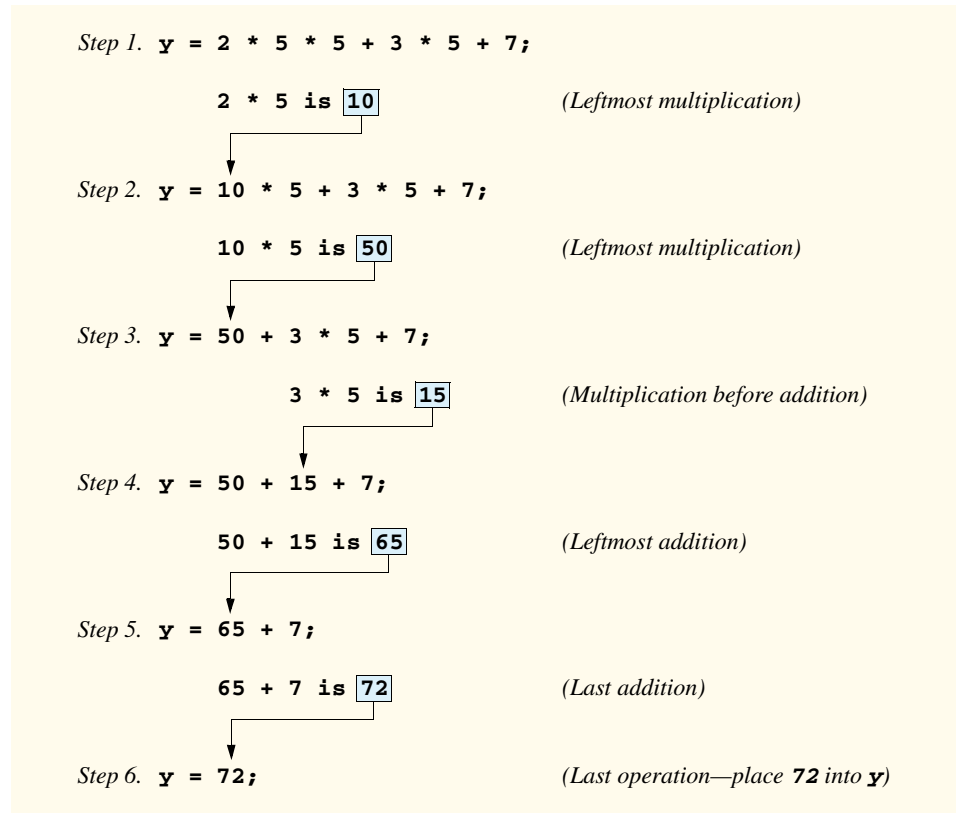


Fig. 2.18 Order in which a second-degree polynomial is evaluated.

Standard algebraic equality or relational operator	Java equality or relational operator	Example of Java condition	Meaning of Java condition
<i>Equality operators</i>			
=	==	$x == y$	x is equal to y
≠	!=	$x != y$	x is not equal to y
<i>Relational operators</i>			
>	>	$x > y$	x is greater than y
<	<	$x < y$	x is less than y
≥	>=	$x >= y$	x is greater than or equal to y
≤	<=	$x <= y$	x is less than or equal to y

Fig. 2.19 Equality and relational operators.



Common Programming Error 2.11

It is a syntax error if the operators `==`, `!=`, `>=` and `<=` contain spaces between their symbols, as in `= =`, `! =`, `> =` and `< =`, respectively.



Common Programming Error 2.12

Reversing the operators `!=`, `>=` and `<=`, as in `=!`, `=>` and `=<`, is a syntax error.



Common Programming Error 2.13

Confusing the equality operator, `==`, with the assignment operator, `=`, can be a logic error or a syntax error. The equality operator should be read as “is equal to,” and the assignment operator should be read as “gets” or “gets the value of.” Some people prefer to read the equality operator as “double equals” or “equals equals.”

The next example uses six **if** structures to compare two numbers input into text fields by the user. If the condition in any of these **if** statements is true, the assignment statement associated with that **if** structure executes. The user inputs two values through input dialogs. Next, the program converts the input values to integers and stores them in variables **number1** and **number2**. Then, the program compares the numbers and displays the results of the comparisons in an information dialog. The program and sample outputs are shown in Fig. 2.20.

```

1  // Fig. 2.20: Comparison.java
2  // Compare integers using if structures, relational operators
3  // and equality operators.
4
5  // Java extension packages
6  import javax.swing.JOptionPane;
7
8  public class Comparison {
9
10     // main method begins execution of Java application
11     public static void main( String args[] )
12     {
13         String firstNumber;    // first string entered by user
14         String secondNumber;   // second string entered by user
15         String result;         // a string containing the output
16         int number1;           // first number to compare
17         int number2;           // second number to compare
18
19         // read first number from user as a string
20         firstNumber =
21             JOptionPane.showInputDialog( "Enter first integer:" );
22
23         // read second number from user as a string
24         secondNumber =
25             JOptionPane.showInputDialog( "Enter second integer:" );
26

```

Fig. 2.20 Using equality and relational operators (part 1 of 3).

```

27      // convert numbers from type String to type int
28      number1 = Integer.parseInt( firstNumber );
29      number2 = Integer.parseInt( secondNumber );
30
31      // initialize result to empty String
32      result = "";
33
34      if ( number1 == number2 )
35          result = number1 + " == " + number2;
36
37      if ( number1 != number2 )
38          result = number1 + " != " + number2;
39
40      if ( number1 < number2 )
41          result = result + "\n" + number1 + " < " + number2;
42
43      if ( number1 > number2 )
44          result = result + "\n" + number1 + " > " + number2;
45
46      if ( number1 <= number2 )
47          result = result + "\n" + number1 + " <= " + number2;
48
49      if ( number1 >= number2 )
50          result = result + "\n" + number1 + " >= " + number2;
51
52      // Display results
53      JOptionPane.showMessageDialog(
54          null, result, "Comparison Results",
55          JOptionPane.INFORMATION_MESSAGE );
56
57      System.exit( 0 ); // terminate application
58
59  } // end method main
60
61 } // end class Comparison

```

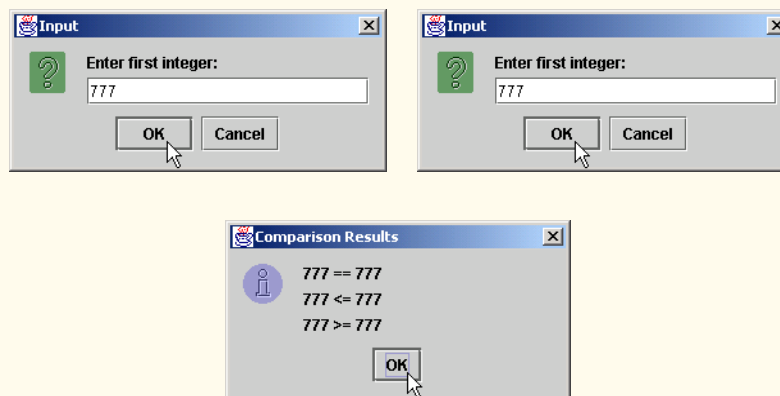


Fig. 2.20 Using equality and relational operators (part 2 of 3).

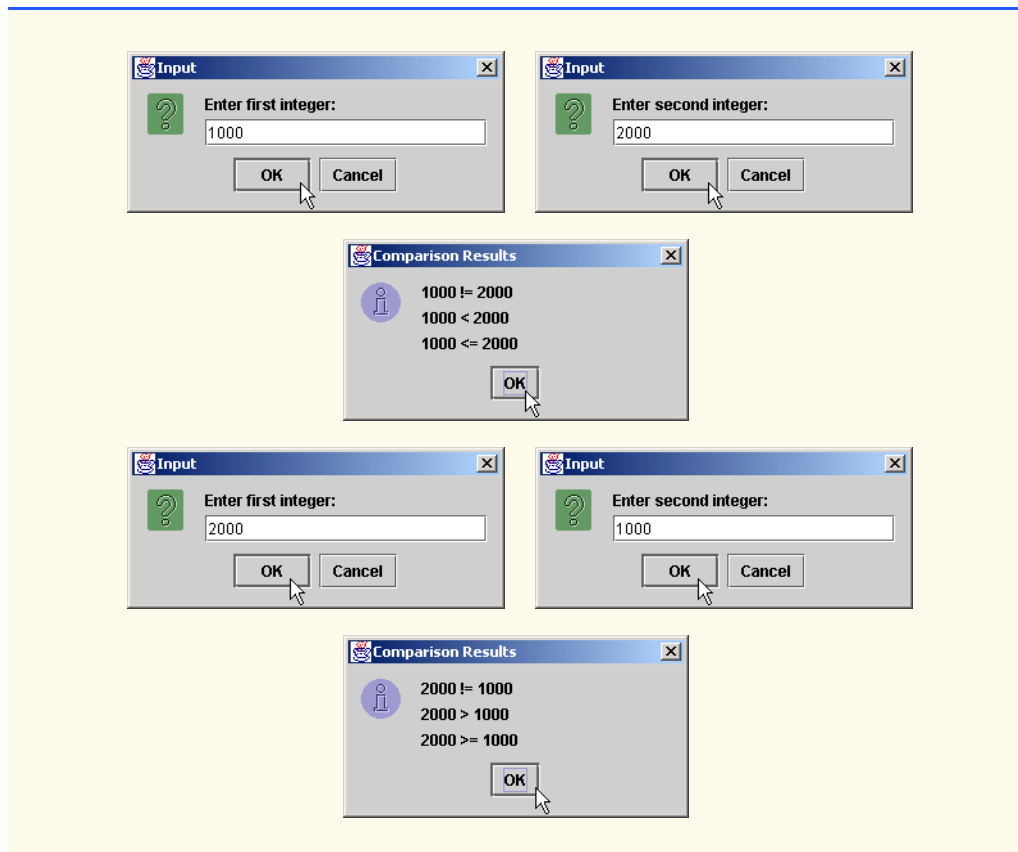


Fig. 2.20 Using equality and relational operators (part 3 of 3).

The definition of application class **Comparison** begins at line 8,

```
public class Comparison {
```

As discussed previously, method **main** (lines 11–59) begins the execution of every Java application.

Lines 13–17,

```
String firstNumber; // first string entered by user
String secondNumber; // second string entered by user
String result; // a string containing the output
int number1; // first number to compare
int number2; // second number to compare
```

declare the variables used in method **main**. Note that there are three variables of type **String** and two variables of type **int**. Remember that variables of the same type may be declared in one declaration or in multiple declarations. If more than one name is declared in a declaration, the names are separated by commas (,), as in

```
String firstNumber, secondNumber, result;
```

or as in

```
String firstNumber,
    secondNumber,
    result;
```

This is set of names known as a *comma-separated list*. Once again, notice the comment at the end of each declaration in lines 13–17, indicating the purpose of each variable in the program.

Lines 20–21,

```
firstNumber =
    JOptionPane.showInputDialog( "Enter first integer:" );
```

use `JOptionPane.showInputDialog` to allow the user to input the first integer value as a string and store it in `firstNumber`.

Lines 24–25,

```
secondNumber =
    JOptionPane.showInputDialog( "Enter second integer:" );
```

use `JOptionPane.showInputDialog` to allow the user to input the second integer value as a string and store it in `secondNumber`.

Lines 28–29,

```
number1 = Integer.parseInt( firstNumber );
number2 = Integer.parseInt( secondNumber );
```

convert each string input by the user in the input dialogs to type `int` and assign the values to int variables `number1` and `number2`.

Line 32,

```
result = "";
```

assigns to `result` the *empty string*—a string containing no characters. Every variable declared in a method (such as `main`) must be *initialized* (given a value) before it can be used in an expression. Because we do not yet know what the final `result` string will be, we assign to `result` the empty string as a temporary initial value.



Common Programming Error 2.14

Not initializing a variable defined in a method before that variable is used in the method's body is a syntax error.

Lines 34–35,

```
if ( number1 == number2 )
    result = result + number1 + " == " + number2;
```

define an *if structure* that compares the values of the variables `number1` and `number2` to determine if they are equal. The *if* structure always begins with keyword `if`, followed by a condition in parentheses. The *if* structure expects one statement in its body. The indentation shown here is not required, but it improves the readability of the program by emphasizing that the statement in line 35 is part of the *if* structure that begins on line 34.



Good Programming Practice 2.17

*Indent the statement in the body of an **if** structure to make the body of the structure stand out and to enhance program readability.*



Good Programming Practice 2.18

Place only one statement per line in a program. This format enhances program readability.

In the preceding **if** structure, if the values of variables **number1** and **number2** are equal, line 35 assigns to **result** the value of **result + number1 + " == " + number2**. As discussed in Fig. 2.9, the **+** operator in this expression performs string concatenation. For this discussion, we assume that each of the variables **number1** and **number2** has the value **123**. First, the expression converts **number1**'s value to a string and appends it to **result** (which currently contains the empty string) to produce the string **"123"**. Next, the expression appends **" == "** to **"123"** to produce the string **"123 == "**. Finally, the expression appends **number2** to **"123 == "** to produce the string **"123 == 123"**. The **String result** becomes longer as the program proceeds through the **if** structures and performs more concatenations. For example, given the value **123** for both **number1** and **number2** in this discussion, the **if** conditions at lines 46–47 (**<=**) and 49–50 (**>=**) are also true. So, the program displays the **result**

```
123 == 123
123 <= 123
123 >= 123
```

in a message dialog.



Common Programming Error 2.15

*Replacing operator **==** in the condition of an **if** structure, such as **if (x == 1)**, with operator **=**, as in **if (x = 1)**, is a syntax error.*



Common Programming Error 2.16

*Forgetting the left and right parentheses for the condition in an **if** structure is a syntax error. The parentheses are required.*

Notice that there is no semicolon (**;**) at the end of the first line of each **if** structure. Such a semicolon would result in a logic error at execution time. For example,

```
if ( number1 == number2 ); // logic error
    result = result + number1 + " == " + number2;
```

would actually be interpreted by Java as

```
if ( number1 == number2 )
;

result = result + number1 + " == " + number2;
```

where the semicolon on the line by itself—called the *empty statement*—is the statement to execute if the condition in the **if** structure is true. When the empty statement executes, no task is performed in the program. The program then continues with the assignment statement, which executes regardless of whether the condition is true or false.



Common Programming Error 2.17

Placing a semicolon immediately after the right parenthesis of the condition in an **if** structure is normally a logic error. The semicolon will cause the body of the **if** structure to be empty, so the **if** structure itself will perform no action, regardless of whether its condition is true. Worse yet, the intended body statement of the **if** structure will now become a statement in sequence with the **if** structure and will always be executed.

Notice the use of spacing in Fig. 2.20. Remember that white-space characters, such as tabs, newlines and spaces, are normally ignored by the compiler. So, statements may be split over several lines and may be spaced according to the programmer's preferences without affecting the meaning of a program. It is incorrect to split identifiers and string literals. Ideally, statements should be kept small, but it is not always possible to do so.



Good Programming Practice 2.19

A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose breaking points that make sense, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines until the end of the statement.

The chart in Fig. 2.21 shows the precedence of the operators introduced in this chapter. The operators are shown from top to bottom in decreasing order of precedence. Notice that all of these operators, with the exception of the assignment operator, **=**, associate from left to right. Addition is left associative, so an expression like **x + y + z** is evaluated as if it had been written as **(x + y) + z**. The assignment operator, **=**, associates from right to left, so an expression like **x = y = 0** is evaluated as if it had been written as **x = (y = 0)**, which, as we will soon see, first assigns the value **0** to variable **y** and then assigns the result of that assignment, **0**, to **x**.



Good Programming Practice 2.20

Refer to the operator precedence chart (see the complete chart in Appendix C) when writing expressions containing many operators. Confirm that the operations in the expression are performed in the order you expect. If you are uncertain about the order of evaluation in a complex expression, use parentheses to force the order, exactly as you would do in algebraic expressions. Be sure to observe that some operators, such as assignment, **=**, associate from right to left rather than from left to right.

Operators	Associativity	Type
()	left to right	parentheses
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
=	right to left	assignment

Fig. 2.21 Precedence and associativity of the operators discussed so far.

We have introduced many important features of Java in this chapter, including displaying data on the screen, inputting data from the keyboard, performing calculations and making decisions. We should note that these applications are meant to introduce the reader to basic programming concepts. As you will see in later chapters, more substantial Java applications contain just a few lines of code in method **main** that creates the objects that perform the work of the application. In Chapter 3, we demonstrate many similar techniques as we introduce Java applet programming. In Chapter 4, we build on the techniques of Chapter 2 and Chapter 3 as we introduce *structured programming*. You will become more familiar with indentation techniques. We will study how to specify and vary the order in which statements are executed; this order is called *flow of control*.

2.9 (Optional Case Study) Thinking About Objects: Examining the Problem Statement

Now we begin our optional, object-oriented design and implementation case study. The “Thinking About Objects” sections at the ends of this and the next several chapters will ease you into object orientation by examining an elevator simulation case study. This case study will provide you with a substantial, carefully paced, complete design and implementation experience. In Chapters 3 through 13, Chapter 15 and Chapter 22, we will perform the various steps of an object-oriented design (OOD) process using the UML while relating to the object-oriented concepts discussed in the chapters. In Appendices G, H and I, we will implement the elevator simulator using the techniques of object-oriented programming (OOP) in Java. We present the complete case-study solution. This is not an exercise; rather, it is an end-to-end learning experience that concludes with a detailed walkthrough of the actual Java code. We have provided this case study so that you can become accustomed to the kinds of substantial problems encountered in industry. We hope you enjoy this learning experience.

Problem Statement

A company intends to build a two-floor office building and equip it with an elevator. The company wants you to develop an object-oriented *software-simulator application* in Java that models the operation of the elevator to determine whether it will meet the company’s needs. The company wants the simulation to contain an elevator system. The application consists of three parts. The first and most substantial part is the simulator, which *models* the operation of the elevator system. The second part is the display of this model on screen so that the user may *view* it graphically. The final part is the *graphical user interface*, or *GUI*, that allows the user to *control* the simulation. Our design and implementation will follow the so-called *Model-View-Controller architecture* we will learn about in Section 13.17.

The elevator system consists of an elevator shaft and an elevator car. In our simulation, we model people who ride the elevator car (referred to as “the elevator”) to travel between the floors in the elevator shaft, as shown in Fig. 2.22, Fig. 2.23 and Fig. 2.24.

The elevator contains a door (called the “elevator door”) that opens upon the elevator’s arrival at a floor and closes upon the elevator’s departure from that floor. The elevator door is closed during the trips between floors to prevent the passenger from being injured by brushing against the wall of the elevator shaft. In addition, the elevator shaft connects to a door on each floor (referred to as the two “floor doors”), so people cannot fall down the shaft when the elevator is not at a floor. Note that we do not display the floor doors in the figures, because they would obscure the inside of the elevator (we use a mesh door to rep-

resent the elevator door because mesh allows us to see inside the elevator). The floor door opens concurrently with the elevator door, so it appears as if both doors open at the same time. A person sees only one door, depending on that person's location. When the person is inside the elevator, the person sees the elevator door and can exit the elevator when this door opens; when the person is outside the elevator, the person sees the floor door and can enter the elevator when that door opens¹.

The elevator starts on the first floor with all the doors closed. To conserve energy, the elevator moves only when necessary. For simplicity, the elevator and floors each have a capacity of only one person.²

The user of our application should, at any time, be able to create a unique person in the simulation and situate that person on either the first or second floor (Fig. 2.22). When created, the person walks across the floor to the elevator. The person then presses a button on the floor next to the elevator shaft (referred to as a "floor button"). When pressed, that floor button illuminates, then requests the elevator. When summoned, the elevator travels to the person's floor. If the elevator is already on that person's floor, the elevator does not travel. Upon arrival, the elevator resets the button inside the elevator (called the "elevator button"), sounds the bell inside the elevator, then opens the elevator door (which opens the floor door on that floor). The elevator then signals the elevator shaft of the arrival. The elevator shaft, upon receiving this message, resets the floor button and illuminates the light on that floor.

Occasionally, a person requests the elevator when it is moving. If the request was generated at the floor from which the elevator just departed, the elevator must "remember" to revisit that floor after carrying the current passenger to the other floor.

When the floor door opens, the person enters the elevator after the elevator passenger (if there is one) exits. If a person neither enters nor requests the elevator, the elevator closes its door and remains on that floor until the next person presses a floor button to summon the elevator.

When a person enters the elevator, that person presses the elevator button, which also illuminates when pressed. The elevator closes its door (which also closes the floor door on that floor) and moves to the opposite floor. The elevator takes five seconds to travel between floors. When the elevator arrives at the destination floor, the elevator door opens (along with the floor door on that floor) and the person exits the elevator.

The application user introduces a person onto the first or second floor by pressing the **First Floor** button or the **Second Floor** button, respectively. When the user presses the **First Floor** button, a person should be created (by the elevator simulation) and positioned on the first floor of the building. When the user presses the **Second Floor** button, a person should be created and positioned on the second floor. Over time, the user can create any number of people in the simulation, but the user cannot create a new person on an occupied floor. For example, Fig. 2.22 shows that the **First Floor** button is disabled to prevent the user from creating more than one person on the first floor. Figure 2.23 shows that this button is reenabled when the person rides the elevator.

-
1. Most people do not consider this when riding an elevator—they really think of one "elevator door," when in reality, there is a door in the elevator and a door on the floor, and these doors open and close in tandem.
 2. After you have studied this case study, you may want to modify it to allow more than one person to ride the elevator at once and more than one person to wait on each floor at once.

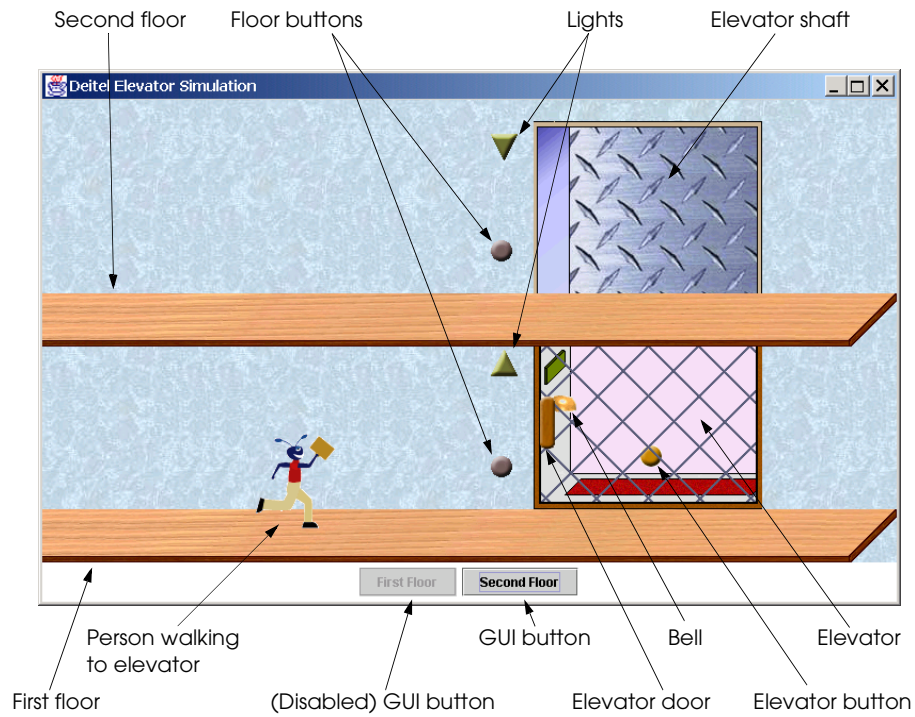


Fig. 2.22 Person moving towards elevator on the first floor.

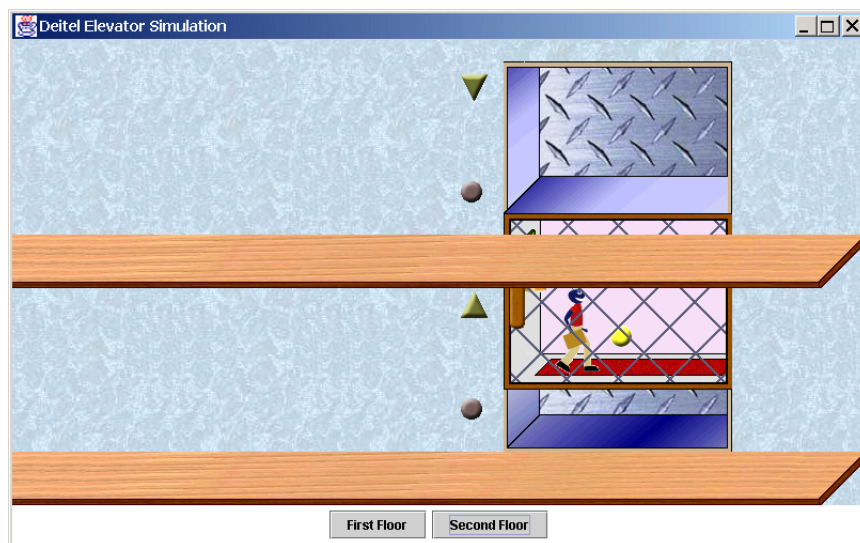


Fig. 2.23 Person riding the elevator to the second floor.

The company requests that we display the results of the simulation graphically, as shown in Fig. 2.22, Fig. 2.23 and Fig. 2.24. At various points in time, the screen should display a person walking to the elevator, pressing a button and entering, riding and exiting the elevator. The display also should show the elevator moving, the doors opening, the lights turning on and off, the buttons illuminating when they are pressed and the buttons darkening when they are reset.

The company requests that audio be integrated into the simulation. For example, as a person walks, the application user should hear the footsteps. Each time a floor or elevator button is pressed or reset, the user should hear a click. The bell should ring upon the elevator's arrival, and doors should creak when they open or close. Lastly, "elevator music" should play as the elevator travels between floors.

Analyzing and Designing the Elevator System

We must analyze and design our system before we implement it as Java code. The output of the analysis phase is intended to specify clearly in a *requirements document* what the system is supposed to do. The requirements document for this case study is essentially the description of what the elevator simulator is supposed to do—presented informally in the last few pages. The output of the design phase should specify clearly *how* the system should be constructed to do what is needed. In the next several "Thinking About Objects" sections, we perform the steps of an object-oriented design (OOD) process on the elevator system. The UML is designed for use with any OOD process—many such processes exist. One popular method is the Rational Unified Process™ developed by Rational Software Corporation. For this case study, we present our own simplified design process. For many of our readers, this will be their first OOD/UML experience.

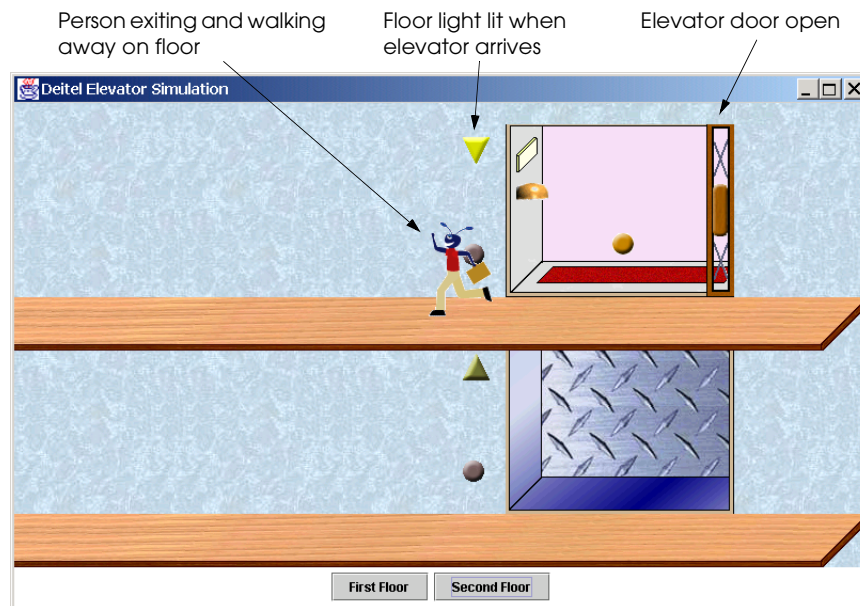


Fig. 2.24 Person walking away from elevator.

We now begin the design phase of our elevator system, which will span Chapters 2 through 13, Chapter 15 and Chapter 22, in which we gradually develop the design. Appendices G, H and I present the complete Java implementation.

A system is a set of components that interact to solve a problem. In our case study, the elevator-simulator application represents the system. A system may contain “subsystems,” which are “systems within a system.” Subsystems simplify the design process by managing subsets of system responsibilities. System designers may allocate system responsibilities among the subsystems, design the subsystems, then integrate the subsystems with the overall system. Our elevator-simulation system contains three subsystems, which are defined in the problem statement:

1. the simulator model (which represents the operation of the elevator system),
2. the display of this model on screen (so that the user may view it graphically), and
3. the graphical user interface (that allows the user to control the simulation).

We develop the simulator model gradually through Chapter 15 and present the implemented model in Appendix H. We discuss the GUI components allowing the user to control the model in Chapter 12 and introduce how the subsystems work together to form the system in Chapter 13. Finally, we introduce how to display the simulator model on the screen in Chapter 22 and conclude the display in Appendix I.

System *structure* describes the system’s objects and their inter-relationships. System *behavior* describes how the system changes as its objects interact with each other. Every system has both structure and behavior—we must design both. However, there are several distinct *types* of system structures and behaviors. For example, the interaction among the objects in the system differs from the interaction between the user and the system, yet both are interactions that constitute the system behavior.

The UML specifies nine types of diagrams for modeling systems. Each diagram models a distinct characteristic of a system’s structure or behavior—the first four diagrams relate to system structure; the remaining five diagrams relate to system behavior:

1. Class diagram
2. Object diagram
3. Component diagram
4. Deployment diagram
5. Activity diagram
6. Statechart diagram
7. Collaboration diagram
8. Sequence diagram
9. Use-Case diagram

Class diagrams, which we explain in “Thinking About Objects” Section 3.8, model the classes, or “building blocks,” used to build a system. Each entity in the problem statement is a candidate to be a class in the system (i.e., **Person**, **Elevator**, **Floor**, etc.).

Object diagrams, which we also explain in Section 3.8, model a “snapshot” of the system by modeling a system’s objects and their relationships at a specific point in time. Each object represents an instance of a class from the class diagram (e.g., the elevator

object is an instance of class **Elevator**), and there may be several objects created from one class (e.g., both the first floor button object and the second floor button object are created from class **FloorButton**).

Component diagrams, presented in Section 13.17, model the *components*—resources (which include graphics and audio) and *packages* (which are groups of classes)—that make up the system.

Deployment diagrams model the runtime requirements of the system (such as the computer or computers on which the system will reside), memory requirements for the system, or other devices the system requires during execution. We do not present deployment diagrams in this case study, because we are not designing a “hardware-specific” system—our simulation requires only one computer containing the Java 2 runtime environment on which to run.

Statechart diagrams, which we introduce in Section 5.11, model *how* an object changes *state* (i.e., the condition of an object at a specific time). When an object changes state, that object may behave differently in the system.

Activity diagrams, which we also introduce in Section 5.11, model an object’s *activity*—that object’s workflow during program execution. An activity diagram is a flowchart that models the actions the object will perform and in what order.

Both *collaboration diagrams* and *sequence diagrams* model the interactions among the objects in a system. Collaboration diagrams emphasize what interactions occur, whereas sequence diagrams emphasize when interactions occur. We introduce these diagrams in Section 7.10 and Section 15.12, respectively.

Use-Case diagrams represent the interaction between the user and our system (i.e., all actions the user may perform on the system). We introduce use-case diagrams in Section 12.16, where we discuss user-interface issues.

In “Thinking About Objects” Section 3.17, we continue designing our elevator system by identifying the classes in the problem statement. We accomplish this by extracting all the nouns and noun clauses from the problem statement. Using these classes, we develop a class diagram that models the structure of our elevator simulation system.

Internet and World-Wide-Web Resources

Listed below are URLs and books on object-oriented design with the UML—you may find these references helpful as you study the remaining sections of our case-study presentation.

www.omg.com/technology/uml/index.htm

This is the UML resource page from the Object Management Group, which provides specifications for various object-oriented technologies, such as the UML.

www.smartdraw.com/drawing/software/indexUML.asp

This site shows how to draw UML diagrams without the use of modeling tools.

www.rational.com/uml/index.jsp

This is the UML resource page for Rational Software Corporation—the company that created the UML.

microgold.com/Stage/UML_FAQ.html

This site provides the UML FAQ maintained by Rational Software.

www.softdocwiz.com/Dictionary.htm

This site hosts the Unified Modeling Language Dictionary, which lists and defines all terms used in the UML.

www.embarcadero.com

This site provides a free 30-day license to download a trial-version of Describe™ — the new UML modeling tool from Embarcadero Technologies®.

www.ics.uci.edu/pub/arch/uml/uml_books_and_tools.html

This site lists books on the UML and software tools that use the UML, such as Rational Rose™ and Embarcadero Describe™.

www.ootips.org/ood-principles.html

This site provides answers to the question “what makes good OOD?”

wdvl.internet.com/Authoring/Scripting/Tutorial/oo_design.html

This site introduces OOD and provides OOD resources.

Bibliography

Booch, G., *Object-Oriented Analysis and Design with Applications*. Addison-Wesley. Massachusetts; 1994.

Folwer, M., and Scott, K., *UML Distilled Second Edition; A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley. Massachusetts; 1999.

Larman, C., *Applying UML and Patterns; An Introduction to Object-Oriented Analysis and Design*. Prentice Hall. New Jersey; 1998.

Page-Jones, M., *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley. Massachusetts; 1999.

Rumbaugh, J.; Jacobson, I.; and Booch, G., *The Unified Modeling Language Reference Manual*. Addison-Wesley. Massachusetts; 1999.

Rumbaugh, J.; Jacobson, I.; and Booch, G., *The Unified Modeling Language User Guide*. Addison-Wesley. Massachusetts; 1999.

Rumbaugh, J.; Jacobson, I.; and Booch, G., *The Complete UML Training Course*. Prentice Hall. New Jersey; 2000.

Rumbaugh, J.; Jacobson, I.; and Booch, G., *The Unified Software Development Process*. Addison-Wesley. Massachusetts; 1999.

Rosenburg, D., and Scott, K., *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Addison-Wesley. Massachusetts; 2001.

Schach, S., *Object-Oriented and Classical Software Engineering*. McGraw Hill. New York; 2001.

Schneider, G., and Winters, J., *Applying Use Cases*. Addison-Wesley. Massachusetts; 1998.

Scott, K., *UML Explained*. Addison-Wesley. Massachusetts; 2001.

Stevens, P., and Pooley, R.J., *Using UML: Software Engineering with Objects and Components Revised Edition*. Addison-Wesley; 2000.

SUMMARY

- An application is a program that executes using the **java** interpreter.
- A comment that begins with **//** is called a single-line comment. Programmers insert comments to document programs and improve program readability.
- A string of characters contained between double quotation marks is called a string, a character string, a message or a string literal.
- Blank lines, space characters, newline characters and tab characters are known as white-space characters. White-space characters outside strings are ignored by the compiler.

- Keyword **class** introduces a class definition and is immediately followed by the class name.
- Keywords (or reserved words) are reserved for use by Java. Keywords must appear in all lower-case letters.
- By convention, all class names in Java begin with a capital letter. If a class name contains more than one word, the first letter of each word should be capitalized.
- An identifier is a series of characters consisting of letters, digits, underscores (`_`) and dollar signs (`$`) that does not begin with a digit, does not contain any spaces and is not a keyword.
- Java is case sensitive—that is, uppercase and lowercase letters are different.
- A left brace, `{`, begins the body of every class definition. A corresponding right brace, `}`, ends each class definition.
- Java applications begin executing at method **main**.
- Methods are able to perform tasks and return information when they complete their tasks.
- The first line of method **main** must be defined as

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

- A left brace, `{`, begins the body of a method definition. A corresponding right brace, `}`, ends the method definition's body.
- **System.out** is known as the standard output object. **System.out** allows Java applications to display strings and other types of information in the command window from which the Java application executes.
- The escape sequence `\n` indicates a newline character. Other escape sequences include `\t` (tab), `\r` (carriage return), `\\` (backslash) and `\"` (double quote).
- Method **println** of the **System.out** object displays (or prints) a line of information in the command window. When **println** completes its task, it automatically positions the output cursor to the beginning of the next line in the command window.
- Every statement must end with a semicolon (also known as the statement terminator).
- The difference between **System.out**'s **print** and **println** methods is that **print** does not position to the beginning of the next line in the command window when it finishes displaying its argument. The next character displayed in the command window appears immediately after the last character displayed with **print**.
- Java contains many predefined classes that are grouped into categories of related classes called packages. The packages are referred to collectively as the Java class library or the Java applications programming interface (Java API).
- Class **JOptionPane** is defined in package **javax.swing**. Class **JOptionPane** contains methods that display dialog boxes.
- The compiler uses **import** statements to locate classes required to compile a Java program.
- The **javax.swing** package contains many classes that help define a graphical user interface (GUI) for an application. GUI components facilitate data entry by the user of a program and data outputs by a program.
- Method **showMessageDialog** of class **JOptionPane** displays a dialog box containing a message to the user.
- A **static** method is called by following its class name by a dot (`.`) and the name of the method.
- Method **exit** of class **System** terminates an application. Class **System** is in package **java.lang**. All Java programs import package **java.lang** by default.

- A variable is a location in the computer's memory where a value can be stored for use by a program. The name of a variable is any valid identifier.
- All variables must be declared with a name and a data type before they can be used in a program.
- Declarations end with a semicolon (`;`) and can be split over several lines, with each variable in the declaration separated by a comma (forming a comma-separated list of variable names).
- Variables of type **int** hold integer values (whole numbers such as 7, -11, 0 and 31,914).
- Types such as **int**, **float**, **double** and **char** are primitive data types. Names of primitive data types are keywords of the Java programming language.
- A prompt directs the user to take a specific action.
- A variable is assigned a value by using an assignment statement, which uses the assignment operator, `=`. The `=` operator is called a binary operator, because it has two operands.
- Method **Integer.parseInt** (a **static** method of class **Integer**) converts its **String** argument to an **int** value.
- Java has a version of the `+` operator for string concatenation that enables a string and a value of another data type (including another string) to be concatenated.
- Every variable has a name, a type, a size and a value.
- When a value is placed in a memory location, the value replaces the value previously in that location. When a value is read out of a memory location, the variable's value remains unchanged.
- The arithmetic operators are binary operators, because they operate on two operands.
- Integer division yields an integer result.
- Arithmetic expressions in Java must be written in straight-line form to facilitate entering programs into the computer.
- Operators in arithmetic expressions are applied in a precise sequence determined by the rules of operator precedence.
- Parentheses may be used to force the order of evaluation of operators.
- When we say that operators are applied from left to right, we are referring to the associativity of the operators. Some operators associate from right to left.
- Java's **if** structure allows a program to make a decision based on the truth or falsity of a condition. If the condition is met (i.e., the condition is true), the statement in the body of the **if** structure executes. If the condition is not met (i.e., the condition is false), the body statement does not execute.
- Conditions in **if** structures can be formed by using the equality operators and relational operators.
- The empty string is a string containing no characters.
- Every variable declared in a method must be initialized before it can be used in an expression.

TERMINOLOGY

addition operator (<code>+</code>)	body of a class definition
applet	body of a method definition
application	braces (<code>{</code> and <code>}</code>)
argument to a method	case sensitive
arithmetic operators	character string
assignment operator (<code>=</code>)	class
assignment statement	class definition
associativity of operators	.class file extension
backslash (<code>\</code>) escape character	class keyword
binary operator	class name

command tool
 command window
 comma-separated list
 comment (*//*)
 compilation error
 compile error
 compiler
 compile-time error
 condition
 decision
 declaration
 dialog
 dialog box
 division operator (*/*)
 document a program
 empty string (*""*)
 equality operators
 == "is equal to"
 != "is not equal to"
 escape sequence
exit method of **System**
 false
 graphical user interface (GUI)
 identifier
if structure
import statement
 input dialog
int primitive type
 integer (**int**)
Integer class
 integer division
 interpreter
 Java
 Java 2 Software Development Kit (J2SDK)
 Java applications programming interface (API)
 Java class library
 Java documentation comment
.java file extension
java interpreter
java.lang package
javax.swing package
JOptionPane class
JOptionPane.ERROR_MESSAGE
JOptionPane.INFORMATION_MESSAGE
JOptionPane.PLAIN_MESSAGE
JOptionPane.QUESTION_MESSAGE
JOptionPane.WARNING_MESSAGE
 left brace, **{**, begins the body of a class
 left brace, **{**, begins the body of a method
 literal
main method
 memory
 memory location
 message
 message dialog
 method
 Microsoft Internet Explorer browser
 modulus operator (*%*)
 mouse cursor
 mouse pointer
 MS-DOS Prompt
 multiple-line comment
 multiplication operator (***)
 nested parentheses
 Netscape Navigator browser
 newline character (*\n*)
 object
 operand
 operator
 package
 parentheses (*()*)
parseInt method of class **Integer**
 precedence
 primitive data type
 programmer-defined class
 prompt
public keyword
 relational operators
 < "is less than"
 <= "is less than or equal to"
 > "is greater than"
 >= "is greater than or equal to"
 reserved words
 right brace, **}**, ends the body of a class
 right brace, **}**, ends the body of a method
 right-to-left associativity
 rules of operator precedence
 semicolon (*;*) statement terminator
 shell tool
showInputDialog method of **JOptionPane**
showMessageDialog method of **JOptionPane**
 single-line comment
 standard output object
 statement
 statement terminator (*;*)
static method
 straight-line form
 string
String class

string concatenation	title bar of a dialog
string concatenation operator (+)	true
subtraction operator (-)	user-defined class
syntax error	variable
System class	variable name
System.out object	variable value
System.out.print method	void keyword
System.out.println method	white-space characters

SELF-REVIEW EXERCISES

2.1 Fill in the blanks in each of the following statements:

- The _____ begins the body of every method, and the _____ ends the body of every method.
- Every statement ends with a _____.
- The _____ structure is used to make decisions.
- _____ begins a single-line comment.
- _____, _____, _____ and _____ are called white-space characters.
- Class _____ contains methods that display message dialogs and input dialogs.
- _____ are reserved for use by Java.
- Java applications begin execution at method _____.
- Methods _____ and _____ display information in the command window.
- A _____ method is always called using its class name followed by a dot (.) and its method name.

2.2 State whether each of the following is *true* or *false*. If *false*, explain why.

- Comments cause the computer to print the text after the `//` on the screen when the program is executed.
- All variables must be given a type when they are declared.
- Java considers the variables **number** and **Number** to be identical.
- The modulus operator (%) can be used only with integer operands.
- The arithmetic operators `*`, `/`, `%`, `+` and `-` all have the same level of precedence.
- Method **Integer.parseInt** converts an integer to a **String**.

2.3 Write Java statements to accomplish each of the following tasks:

- Declare variables **c**, **thisIsAVariable**, **q76354** and **number** to be of type **int**.
- Display a dialog asking the user to enter an integer.
- Convert a **String** to an integer, and store the converted value in integer variable **age**. Assume that the **String** is stored in **value**.
- If the variable **number** is not equal to 7, display "The variable number is not equal to 7" in a message dialog. [Hint: Use the version of the message dialog that requires two arguments.]
- Print the message "This is a Java program" on one line in the command window.
- Print the message "This is a Java program" on two lines in the command window; the first line should end with **Java**. Use only one statement.

2.4 Identify and correct the errors in each of the following statements:

- ```
if (c < 7);
 JOptionPane.showMessageDialog(null,
 "c is less than 7");
```
- ```
if ( c => 7 )
    JOptionPane.showMessageDialog( null,
        "c is equal to or greater than 7" );
```

- 2.5** Write a statement (or comment) to accomplish each of the following tasks:
- State that a program will calculate the product of three integers.
 - Declare the variables **x**, **y**, **z** and **result** to be of type **int**.
 - Declare the variables **xVal**, **yVal** and **zVal** to be of type **String**.
 - Prompt the user to enter the first value, read the value from the user and store it in the variable **xVal**.
 - Prompt the user to enter the second value, read the value from the user and store it in the variable **yVal**.
 - Prompt the user to enter the third value, read the value from the user and store it in the variable **zVal**.
 - Convert **xVal** to an **int**, and store the result in the variable **x**.
 - Convert **yVal** to an **int**, and store the result in the variable **y**.
 - Convert **zVal** to an **int**, and store the result in the variable **z**.
 - Compute the product of the three integers contained in variables **x**, **y** and **z**, and assign the result to the variable **result**.
 - Display a dialog containing the message "The product is " followed by the value of the variable **result**.
 - Return a value from the program indicating that the program terminated successfully.
- 2.6** Using the statements you wrote in Exercise 2.5, write a complete program that calculates and prints the product of three integers.

ANSWERS TO SELF-REVIEW EXERCISES

- 2.1** a) left brace (**{**), right brace (**}**). b) semicolon (**;**). c) **if**. d) **//**. e) Blank lines, space characters, newline characters and tab characters. f) **JOptionPane**. g) Keywords. h) **main**. i) **System.out.print** and **System.out.println**. j) **static**.
- 2.2** a) False. Comments do not cause any action to be performed when the program is executed. They are used to document programs and improve their readability.
 b) True.
 c) False. Java is case sensitive, so these variables are distinct.
 d) False. The modulus operator can also be used with noninteger operands in Java.
 e) False. The operators *****, **/** and **%** are on the same level of precedence, and the operators **+** and **-** are on a lower level of precedence.
 f) False. **Integer.parseInt** method converts a **String** to an integer (**int**) value.
- 2.3** a) **int c, thisIsAVariable, q76354, number;**
 b) **value = JOptionPane.showInputDialog("Enter an integer");**
 c) **age = Integer.parseInt(value);**
 d) **if (number != 7)**
 JOptionPane.showMessageDialog(null,
 "The variable number is not equal to 7");
 e) **System.out.println("This is a Java program");**
 f) **System.out.println("This is a Java\nprogram");**
- 2.4** The solutions to Self-Review Exercise 2.4 are as follows:
 a) Error: Semicolon after the right parenthesis of the condition in the **if** statement.
 Correction: Remove the semicolon after the right parenthesis. [Note: The result of this error is that the output statement will be executed regardless of whether the condition in the **if** statement is true. The semicolon after the right parenthesis is considered an empty statement—a statement that does nothing. We will learn more about the empty statement in the next chapter.]

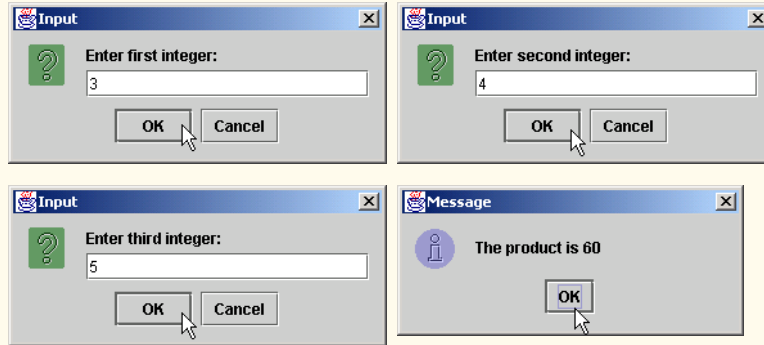
- b) Error: The relational operator `=>` is incorrect.
Correction: Change `=>` to `>=`.

2.5 a) `// Calculate the product of three integers`

- b) `int x, y, z, result;`
c) `String xVal, yVal, zVal;`
d) `xVal = JOptionPane.showInputDialog(`
 `"Enter first integer:" );`
e) `yVal = JOptionPane.showInputDialog(`
 `"Enter second integer:" );`
f) `zVal = JOptionPane.showInputDialog(`
 `"Enter third integer:" );`
g) `x = Integer.parseInt( xVal );`
h) `y = Integer.parseInt( yVal );`
i) `z = Integer.parseInt( zVal );`
j) `result = x * y * z;`
k) `JOptionPane.showMessageDialog( null,`
 `"The product is " + result );`
l) `System.exit( 0 );`

2.6 The solution to Exercise 2.6 is as follows:

```
1  // Calculate the product of three integers
2
3  // Java extension packages
4  import javax.swing.JOptionPane;
5
6  public class Product {
7
8      public static void main( String args[] )
9      {
10         int x, y, z, result;
11         String xVal, yVal, zVal;
12
13         xVal = JOptionPane.showInputDialog(
14             "Enter first integer:" );
15         yVal = JOptionPane.showInputDialog(
16             "Enter second integer:" );
17         zVal = JOptionPane.showInputDialog(
18             "Enter third integer:" );
19
20         x = Integer.parseInt( xVal );
21         y = Integer.parseInt( yVal );
22         z = Integer.parseInt( zVal );
23
24         result = x * y * z;
25         JOptionPane.showMessageDialog( null,
26             "The product is " + result );
27
28         System.exit( 0 );
29     }
30 }
```



EXERCISES

- 2.7** Fill in the blanks in each of the following statements:
- _____ are used to document a program and improve its readability.
 - An input dialog capable of receiving input from the user is displayed with method _____ of class _____.
 - A decision can be made in a Java program with an _____.
 - Calculations are normally performed by _____ statements.
 - A dialog capable of displaying a message to the user is displayed with method _____ of class _____.
- 2.8** Write Java statements that accomplish each of the following tasks:
- Display the message **"Enter two numbers"**, using class **JOptionPane**.
 - Assign the product of variables **b** and **c** to variable **a**.
 - State that a program performs a sample payroll calculation (i.e., use text that helps to document a program).
- 2.9** State whether each of the following is *true* or *false*. If *false*, explain why.
- Java operators are evaluated from left to right.
 - The following are all valid variable names: **_under_bar_**, **m928134**, **t5**, **j7**, **her_sales\$**, **his_\$account_total**, **a**, **b\$**, **c**, **z**, **z2**.
 - A valid Java arithmetic expression with no parentheses is evaluated from left to right.
 - The following are all invalid variable names: **3g**, **87**, **67h2**, **h22**, **2h**.
- 2.10** Fill in the blanks in each of the following statements:
- What arithmetic operations have the same precedence as multiplication? _____.
 - When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression? _____.
 - A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a _____.
- 2.11** What displays in the message dialog when each of the given Java statements is performed? Assume that **x = 2** and **y = 3**.
- `JOptionPane.showMessageDialog( null, "x = " + x );`
 - `JOptionPane.showMessageDialog( null, "The value of x + x is " + ( x + x ) );`
 - `JOptionPane.showMessageDialog( null, "x =" );`
 - `JOptionPane.showMessageDialog( null, ( x + y ) + " = " + ( y + x ) );`

2.12 Which of the following Java statements contain variables whose values are changed or replaced?

- a) `p = i + j + k + 7;`
- b) `JOptionPane.showMessageDialog( null, "variables whose values are destroyed" );`
- c) `JOptionPane.showMessageDialog( null, "a = 5" );`
- d) `stringVal = JOptionPane.showInputDialog( "Enter string: " );`

2.13 Given that $y = ax^3 + 7$, which of the following are correct Java statements for this equation?

- a) `y = a * x * x * x + 7;`
- b) `y = a * x * x * ( x + 7 );`
- c) `y = ( a * x ) * x * ( x + 7 );`
- d) `y = ( a * x ) * x * x + 7;`
- e) `y = a * ( x * x * x ) + 7;`
- f) `y = a * x * ( x * x + 7 );`

2.14 State the order of evaluation of the operators in each of the following Java statements, and show the value of `x` after each statement is performed:

- a) `x = 7 + 3 * 6 / 2 - 1;`
- b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
- c) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`

2.15 Write an application that displays the numbers 1 to 4 on the same line, with each pair of adjacent numbers separated by one space. Write the program using the following methods:

- a) Using one `System.out` statement.
- b) Using four `System.out` statements.

2.16 Write an application that asks the user to enter two numbers, obtains the numbers from the user and prints the sum, product, difference and quotient (division) of the numbers. Use the techniques shown in Fig. 2.9.

2.17 Write an application that asks the user to enter two integers, obtains the numbers from the user and displays the larger number followed by the words “**is larger**” in an information message dialog. If the numbers are equal, print the message “**These numbers are equal.**” Use the techniques shown in Fig. 2.20.

2.18 Write an application that inputs three integers from the user and displays the sum, average, product, smallest and largest of the numbers in an information message dialog. Use the GUI techniques shown in Fig. 2.20. [Note: The calculation of the average in this exercise should result in an integer representation of the average. So, if the sum of the values is 7, the average should be 2, not 2.3333....]

2.19 Write an application that inputs from the user the radius of a circle and prints the circle’s diameter, circumference and area. Use the value 3.14159 for π . Use the GUI techniques shown in Fig. 2.9. [Note: You may also use the predefined constant `Math.PI` for the value of π . This constant is more precise than the value 3.14159. Class `Math` is defined in the `java.lang` package, so you do not need to `import` it.] Use the following formulas (r is the radius):

$$\begin{aligned} \text{diameter} &= 2r \\ \text{circumference} &= 2\pi r \\ \text{area} &= \pi r^2 \end{aligned}$$

2.20 Write an application that displays in the command window a box, an oval, an arrow and a diamond using asterisks (*), as follows:



2.21 Modify the program you created in Exercise 2.20 to display the shapes in a **JOptionPane.PLAIN_MESSAGE** dialog. Does the program display the shapes exactly as in Exercise 2.20?

2.22 What does the following code print?

```
System.out.println( " *\n*\n*\n*\n*\n*\n*\n*\n*\n*" );
```

2.23 What does the following code print?

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

2.24 What does the following code print?

```
System.out.print( " *" );
System.out.print( " ****" );
System.out.print( " *****" );
System.out.print( " *****" );
System.out.println( " ****" );
```

2.25 What does the following code print?

```
System.out.print( " *" );
System.out.println( " ****" );
System.out.println( " *****" );
System.out.print( " *****" );
System.out.println( " ****" );
```

2.26 Write an application that reads five integers and determines and prints the largest and the smallest integers in the group. Use only the programming techniques you learned in this chapter.

2.27 Write an application that reads an integer and determines and prints whether it is odd or even. [Hint: Use the modulus operator. An even number is a multiple of 2. Any multiple of 2 leaves a remainder of 0 when divided by 2.]

2.28 Write an application that reads in two integers and determines and prints if the first is a multiple of the second. [Hint: Use the modulus operator.]

2.29 Write an application that displays in the command window a checkerboard pattern as follows:

```

* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *

```

2.30 Modify the program you wrote in Exercise 2.29 to display the checkerboard pattern in a `JOptionPane.PLAIN_MESSAGE` dialog. Does the program display the shapes exactly as in Exercise 2.29?

2.31 Here's a peek ahead. In this chapter, you have learned about integers and the data type `int`. Java can also represent uppercase letters, lowercase letters and a considerable variety of special symbols. Every character has a corresponding integer representation. The set of characters a computer uses and the corresponding integer representations for those characters is called that computer's *character set*. You can indicate a character value in a program simply by enclosing that character in single quotes, as in `'A'`.

You can determine the integer equivalent of a character by preceding that character with `(int)`. This form is called a cast (we will say more about casts in Chapter 4) as in:

```
(int) 'A'
```

The following statement outputs a character and its integer equivalent:

```
System.out.println( "The character " + 'A' +
    " has the value " + ( int ) 'A' );
```

When the preceding statement executes, it displays the character **A** and the value **65** (from the so-called Unicode character set) as part of the string.

Write an application that displays the integer equivalents of some uppercase letters, lowercase letters, digits and special symbols. At a minimum, display the integer equivalents of the following: **A B C a b c 0 1 2 \$ * + /** and the blank character.

2.32 Write an application that inputs one number consisting of five digits from the user, separates the number into its individual digits and prints the digits separated from one another by three spaces each. For example, if the user types in the number **42339**, the program should print

```
4    2    3    3    9
```

[Hint: It is possible to do this exercise with the techniques you learned in this chapter. You will need to use both division and modulus operations to “pick off” each digit.]

Assume that the user enters the correct number of digits. What happens when you execute the program and type a number with more than five digits? What happens when you execute the program and type a number with fewer than five digits?

2.33 Using only the programming techniques you learned in this chapter, write an application that calculates the squares and cubes of the numbers from 0 to 10 and prints the resulting values in table format as follows:

number	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

[*Note:* This program does not require any input from the user.]

2.34 Write a program that reads a first name and a last name from the user as two separate inputs and concatenates the first name and last name, separating them by a space. Display in a message dialog the concatenated name.

2.35 Write a program that inputs five numbers and determines and prints the number of negative numbers input, the number of positive numbers input and the number of zeros input.