2

Introduction to Java Applications; Input/Output and Operators

What's in a name?
That which we call a rose
By any other name would smell as sweet.
—William Shakespeare

The chief merit of language is clearness.
—Galen

One person can make a difference and every person should try.
—John F. Kennedy

Objectives

In this chapter you'll:

- Write simple Java applications.
- Use input and output statements.
- Learn about Java's primitive types.
- Understand basic memory concepts.
- Use arithmetic operators.
- Learn the precedence of arithmetic operators.
- Write decision-making statements.
- Use relational and equality operators.

Outline

2.1 Introduction

This chapter introduces Java application programming. We begin with examples of programs that display (output) messages on the screen. We then present a program that obtains (inputs) two numbers from a user, calculates their sum and displays the result. You'll learn how to instruct the computer to perform arithmetic calculations and save their results for later use. The last example demonstrates how to make decisions. The application compares two numbers, then displays messages that show the comparison results. You'll use the JDK command-line tools to compile and run this chapter's programs. If you prefer to use an integrated development environment (IDE), we've also posted Dive Into® videos at `http://www.deitel.com/books/jhtp10LOV/` for Eclipse, NetBeans and IntelliJ IDEA.

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

A Java **application** is a computer program that executes when you use the **`java` command** to launch the Java Virtual Machine (JVM). Later in this section we'll discuss how to compile and run a Java application. First we consider a simple application that displays a line of text. Figure 2.1 shows the program followed by a box that displays its output.

```
1  // Fig. 2.1: Welcome1.java
2  // Text-printing program.
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 } // end class Welcome1
```

Fig. 2.1 | Text-printing program. (Part 1 of 2.)

```
Welcome to Java Programming!
```

Fig. 2.1 | Text-printing program. (Part 2 of 2.)

The program includes line numbers. We've added these for instructional purposes—they're *not* part of a Java program. This example illustrates several important Java features. We'll see that line 9 does the work—displaying the phrase `Welcome to Java Programming!` on the screen.

Commenting Your Programs

We insert **comments** to **document programs** and improve their readability. The Java compiler *ignores* comments, so they do *not* cause the computer to perform any action when the program is run.

By convention, we begin every program with a comment indicating the figure number and filename. The comment in line 1

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

begins with **//**, indicating that it's an **end-of-line comment**—it terminates at the end of the line on which the // appears. An end-of-line comment need not begin a line; it also can begin in the middle of a line and continue until the end (as in lines 6, 10 and 11). Line 2

```
// Text-printing program.
```

by our convention, is a comment that describes the purpose of the program.

Java also has **traditional comments**, which can be spread over several lines as in

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

These begin and end with delimiters, **/*** and ***/**. The compiler ignores all text between the delimiters. Java incorporated traditional comments and end-of-line comments from the C and C++ programming languages, respectively. We prefer using // comments.

Java provides comments of a third type—**Javadoc comments**. These are delimited by **/**** and ***/**. The compiler ignores all text between the delimiters. Javadoc comments enable you to embed program documentation directly in your programs. Such comments are the preferred Java documenting format in industry. The **javadoc utility program** (part of the JDK) reads Javadoc comments and uses them to prepare program documentation in HTML format. We demonstrate Javadoc comments and the `javadoc` utility in online Appendix G, Creating Documentation with `javadoc`.

Common Programming Error 2.1

Forgetting one of the delimiters of a traditional or Javadoc comment is a syntax error. A ***syntax error*** *occurs when the compiler encounters code that violates Java's language rules (i.e., its syntax). These rules are similar to a natural language's grammar rules specifying sentence structure. Syntax errors are also called* ***compiler errors***, ***compile-time errors*** *or* ***compilation errors***, *because the compiler detects them when compiling the program. When a syntax error is encountered, the compiler issues an error message. You must eliminate all compilation errors before your program will compile properly.*

Good Programming Practice 2.1

Some organizations require that every program begin with a comment that states the purpose of the program and the author, date and time when the program was last modified.

Using Blank Lines

Line 3 is a blank line. Blank lines, space characters and tabs make programs easier to read. Together, they're known as **white space** (or whitespace). The compiler ignores white space.

Good Programming Practice 2.2

Use blank lines and spaces to enhance program readability.

Declaring a Class

Line 4

```
public class Welcome1
```

begins a **class declaration** for class `Welcome1`. Every Java program consists of at least one class that you (the programmer) define. The **`class` keyword** introduces a class declaration and is immediately followed by the **class name** (`Welcome1`). **Keywords** (sometimes called **reserved words**) are reserved for use by Java and are always spelled with all lowercase letters. The complete list of keywords is shown in Appendix C.

In Chapters 1–7, every class we define begins with the **`public`** keyword. For now, we simply require `public`. You'll learn more about `public` and non-`public` classes in Chapter 8.

Filename for a `public` Class

A `public` class *must* be placed in a file that has a filename of the form *ClassName*`.java`, so class `Welcome1` is stored in the file `Welcome1.java`.

Common Programming Error 2.2

A compilation error occurs if a `public` class's filename is not exactly same name as the class (in terms of both spelling and capitalization) followed by the `.java` extension.

Class Names and Identifiers

By convention, class names begin with a capital letter and capitalize the first letter of each word they include (e.g., `SampleClassName`). A class name is an **identifier**—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. Normally, an identifier that does not begin with a capital letter is not a class name. Java is **case sensitive**—uppercase and lowercase letters are distinct—so `value` and `Value` are different (but both valid) identifiers.

Class Body

A **left brace** (as in line 5), **`{`**, begins the **body** of every class declaration. A corresponding **right brace** (at line 11), **`}`**, must end each class declaration. Lines 6–10 are indented.

Good Programming Practice 2.3

Indent the entire body of each class declaration one "level" between the left brace and the right brace that delimit the body of the class. This format emphasizes the class declaration's structure and makes it easier to read. We use three spaces to form a level of indent—many programmers prefer two or four spaces. Whatever you choose, use it consistently.

Error-Prevention Tip 2.1

When you type an opening left brace, `{`, immediately type the closing right brace, `}`, then reposition the cursor between the braces and indent to begin typing the body. This practice helps prevent errors due to missing braces. Many IDEs insert the closing right brace for you when you type the opening left brace.

Common Programming Error 2.3

It's a syntax error if braces do not occur in matching pairs.

Good Programming Practice 2.4

IDEs typically indent code for you. The Tab *key may also be used to indent code. You can configure each IDE to specify the number of spaces inserted when you press* Tab.

Declaring a Method

Line 6

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

is an end-of-line comment indicating the purpose of lines 7–10 of the program. Line 7

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

is the starting point of every Java application. The **parentheses** after the identifier `main` indicate that it's a program building block called a **method**. Java class declarations normally contain one or more methods. For a Java application, one of the methods *must* be called `main` and must be defined as shown in line 7; otherwise, the Java Virtual Machine (JVM) will not execute the application. Methods perform tasks and can return information when they complete their tasks. We'll explain the purpose of keyword `static` in Chapter 5. Keyword **`void`** indicates that this method will *not* return any information. Later, we'll see how a method can return information. For now, simply mimic `main`'s first line in your Java applications. In line 7, the `String[] args` in parentheses is a required part of the method `main`'s declaration—we discuss this in Chapter 6.

The left brace in line 8 begins the **body of the method declaration**. A corresponding right brace must end it (line 10). Line 9 in the method body is indented between the braces.

Good Programming Practice 2.5

Indent the entire body of each method declaration one "level" between the braces that define the body of the method. This makes the structure of the method stand out and makes the method declaration easier to read.

Performing Output with `System.out.println`

Line 9

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

instructs the computer to perform an action—namely, to display the characters contained between the double quotation marks (the quotation marks themselves are *not* displayed). Together, the quotation marks and the characters between them are a **string**—also known as a **character string** or a **string literal**. White-space characters in strings are *not* ignored by the compiler. Strings *cannot* span multiple lines of code.

The `System.out` object—which is predefined for you—is known as the **standard output object**. It allows a Java application to display information in the **command window** from which it executes. In recent versions of Microsoft Windows, the command window is the Command Prompt. In UNIX/Linux/Mac OS X, the command window is called a **terminal window** or a **shell**. Many programmers call it simply the **command line**.

Method `System.out.println` displays (or prints) a line of text in the command window. The string in the parentheses in line 9 is the **argument** to the method. When `System.out.println` completes its task, it positions the output cursor (the location where the next character will be displayed) at the beginning of the next line in the command window. This is similar to what happens when you press the *Enter* key while typing in a text editor—the cursor appears at the beginning of the next line in the document.

The entire line 9, including `System.out.println`, the argument `"Welcome to Java Programming!"` in the parentheses and the **semicolon** (`;`), is called a **statement**. A method typically contains one or more statements that perform its task. Most statements end with a semicolon. When the statement in line 9 executes, it displays `Welcome to Java Programming!` in the command window.

When learning how to program, sometimes it's helpful to "break" a working program so you can familiarize yourself with the compiler's syntax-error messages. These messages do not always state the exact problem in the code. When you encounter an error message, it will give you an idea of what caused the error. [Try removing a semicolon or brace from the program of Fig. 2.1, then recompile the program to see the error messages generated by the omission.]

Error-Prevention Tip 2.2

When the compiler reports a syntax error, it may not be on the line that the error message indicates. First, check the line for which the error was reported. If you don't find an error on that line, check several preceding lines.

Using End-of-Line Comments on Right Braces for Readability

As an aid to programming novices, we include an end-of-line comment after a closing brace that ends a method declaration and after a closing brace that ends a class declaration. For example, line 10

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

indicates the closing brace of method `main`, and line 11

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

indicates the closing brace of class `Welcome1`. Each comment indicates the method or class that the right brace terminates. We'll omit such ending comments after this chapter.

Compiling Your First Java Application

We're now ready to compile and execute our program. We assume you're using the Java Development Kit's command-line tools, not an IDE. To help you compile and run your

programs in an IDE, we provide online Dive Into® videos for the popular IDEs Eclipse, NetBeans and IntelliJ IDEA. These are located on the book's website:

```
http://www.deitel.com/books/jhtp10LOV
```

To prepare to compile the program, open a command window and change to the directory where the program is stored. Many operating systems use the command `cd` to change directories. On Windows, for example,

```
cd c:\examples\ch02\fig02_01
```

changes to the `fig02_01` directory. On UNIX/Linux/Max OS X, the command

```
cd ~/examples/ch02/fig02_01
```

changes to the `fig02_01` directory. To compile the program, type

```
javac Welcome1.java
```

If the program contains no compilation errors, this command creates a new file called `Welcome1.class` (known as the **class file** for `Welcome1`) containing the platform-independent Java bytecodes that represent our application. When we use the `java` command to execute the application on a given platform, the JVM will translate these bytecodes into instructions that are understood by the underlying operating system and hardware.

Common Programming Error 2.4

When using `javac`, if you receive a message such as "`bad command or filename`," "`javac: command not found`" or "`'javac' is not recognized as an internal or external command, operable program or batch file`," then your Java software installation was not completed properly. This indicates that the system's `PATH` environment variable was not set properly. Carefully review the installation instructions in the Before You Begin section of this book. On some systems, after correcting the `PATH`, you may need to reboot your computer or open a new command window for these settings to take effect.

Each syntax-error message contains the filename and line number where the error occurred. For example, `Welcome1.java:6` indicates that an error occurred at line 6 in `Welcome1.java`. The rest of the message provides information about the syntax error.

Common Programming Error 2.5

The compiler error message "`class Welcome1 is public, should be declared in a file named Welcome1.java`" indicates that the filename does not match the name of the `public` class in the file or that you typed the class name incorrectly when compiling the class.

Executing the `Welcome1` Application

The following instructions assume that the book's examples are located in `C:\examples` on Windows or in your user account's `Documents/examples` folder on Linux/OS X. To execute this program in a command window, change to the directory containing `Welcome1.java`—`C:\examples\ch02\fig02_01` on Microsoft Windows or `~/Documents/examples/ch02/fig02_01` on Linux/OS X. Next, type

```
java Welcome1
```

and press *Enter*. This command launches the JVM, which loads the `Welcome1.class` file. The command *omits* the `.class` file-name extension; otherwise, the JVM will *not* execute

the program. The JVM calls class `Welcome1`'s `main` method. Next, the statement at line 9 of `main` displays `"Welcome to Java Programming!"`. Figure 2.2 shows the program executing in a Microsoft Windows **Command Prompt** window. [*Note:* Many environments show command windows with black backgrounds and white text. We adjusted these settings to make our screen captures more readable.]

Error-Prevention Tip 2.3

When attempting to run a Java program, if you receive a message such as "`Exception in thread "main" java.lang.NoClassDefFoundError: Welcome1`," your `CLASSPATH` environment variable has not been set properly. Please carefully review the installation instructions in the Before You Begin section of this book. On some systems, you may need to reboot your computer or open a new command window after configuring the `CLASSPATH`.

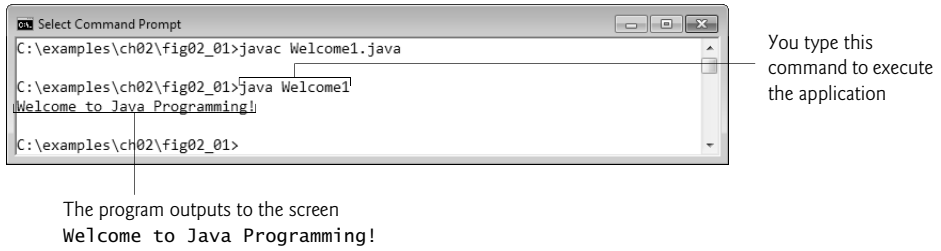


Fig. 2.2 | Executing `Welcome1` from the **Command Prompt**.

2.3 Modifying Your First Java Program

In this section, we 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.

Displaying a Single Line of Text with Multiple Statements

`Welcome to Java Programming!` can be displayed several ways. Class `Welcome2`, shown in Fig. 2.3, uses two statements (lines 9–10) to produce the output shown in Fig. 2.1. [*Note:* From this point forward, we highlight with a yellow background the new and key features in each code listing, as we've done for lines 9–10.]

```
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    } // end method main
12 } // end class Welcome2
```

Fig. 2.3 | Printing a line of text with multiple statements. (Part 1 of 2.)

```
Welcome to Java Programming!
```

Fig. 2.3 | Printing a line of text with multiple statements. (Part 2 of 2.)

The program is similar to Fig. 2.1, so we discuss only the changes here. Line 2

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

is an end-of-line comment stating the purpose of the program. Line 4 begins the `Welcome2` class declaration. Lines 9–10 of method `main`

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

display one line of text. The first statement uses `System.out`'s method `print` to display a string. Each `print` or `println` statement resumes displaying characters from where the last `print` or `println` statement stopped displaying characters. Unlike `println`, 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 will appear *immediately after* the last character that `print` displays. So, line 10 positions the first character in its argument (the letter "`J`") immediately after the last character that line 9 displays (the *space character* before the string's closing double-quote character).

Displaying Multiple Lines of Text with a Single Statement

A single statement can display multiple lines by using **newline characters**, which indicate to `System.out`'s `print` and `println` methods when to position the output cursor at the beginning of the next line in the command window. Like blank lines, space characters and tab characters, newline characters are whitespace characters. The program in Fig. 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 in Figs. 2.1 and 2.3.

```
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\nto\nJava\nProgramming!");
10    } // end method main
11 } // end class Welcome3
```

```
Welcome
to
Java
Programming!
```

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

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. Note, however, that the paired characters \ and n (repeated three times in the statement) do *not* appear on the screen. The **backslash** (\) is an **escape character**, which has special meaning to System.out's print and println methods. When a backslash appears in a string, Java combines it with the next character to form an **escape sequence**. The escape sequence \n represents 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.

Figure 2.5 lists several common escape sequences and describes how they affect the display of characters in the command window. For the complete list of escape sequences, visit

```
http://docs.oracle.com/javase/specs/jls/se7/html/
   jls-3.html#jls-3.10.6
```

Escape sequence	Description
\n	Newline. Position the screen cursor at the beginning of the *next* line.
\t	Horizontal tab. Move the screen cursor to the next tab stop.
\r	Carriage return. Position the screen cursor at 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.
\\	Backslash. Used to print a backslash character.
\"	Double quote. Used to print a double-quote character. For example, `System.out.println("\"in quotes\"");` displays "in quotes".

Fig. 2.5 | Some common escape sequences.

2.4 Displaying Text with printf

The **System.out.printf** method (f means "formatted") displays *formatted* data. Figure 2.6 uses this method to output on two lines the strings "Welcome to" and "Java Programming!". Lines 9–10

```
System.out.printf("%s%n%s%n",
   "Welcome to", "Java Programming!");
```

call method System.out.printf to display the program's output. The method call specifies three arguments. When a method requires multiple arguments, they're placed in a **comma-separated list**.

Good Programming Practice 2.6

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

```
1  // Fig. 2.6: Welcome4.java
2  // Displaying multiple lines with method System.out.printf.
3
4  public class Welcome4
5  {
6     // main method begins execution of Java application
7     public static void main(String[] args)
8     {
9        System.out.printf("%s%n%s%n",
10          "Welcome to", "Java Programming!");
11    } // end method main
12 } // end class Welcome4
```

```
Welcome to
Java Programming!
```

Fig. 2.6 | Displaying multiple lines with method `System.out.printf`.

Lines 9–10 represent only *one* statement. Java allows large statements to be split over many lines. We indent line 10 to indicate that it's a *continuation* of line 9.

Common Programming Error 2.6

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

Method `printf`'s first argument is a **format string** that may consist of **fixed text** and **format specifiers**. Fixed text is output by `printf` just as it would be by `print` or `println`. Each format specifier is a *placeholder* for a value and specifies the *type of data* to output. Format specifiers also may include optional formatting information.

Format specifiers begin with a percent sign (%) followed by a character that represents the *data type*. For example, the format specifier **`%s`** is a placeholder for a string. The format string in line 9 specifies that `printf` should output two strings, each followed by a newline character. At the first format specifier's position, `printf` substitutes the value of the first argument after the format string. At each subsequent format specifier's position, `printf` substitutes the value of the next argument. So this example substitutes `"Welcome to"` for the first `%s` and `"Java Programming!"` for the second `%s`. The output shows that two lines of text are displayed on two lines.

Notice that instead of using the escape sequence `\n`, we used the **`%n`** format specifier, which is a line separator that's *portable* across operating systems. You cannot use `%n` in the argument to `System.out.print` or `System.out.println`; however, the line separator output by `System.out.println` *after* it displays its argument *is* portable across operating systems. Online Appendix I presents more details of formatting output with `printf`.

2.5 Another Application: Adding Integers

Our next application reads (or inputs) two **integers** (whole numbers, such as –22, 7, 0 and 1024) typed by a user at the keyboard, computes their sum and displays it. This program must keep track of the numbers supplied by the user for the calculation later in the program. Programs remember numbers and other data in the computer's memory and access

that data through program elements called **variables**. The program of Fig. 2.7 demonstrates these concepts. In the sample output, we use bold text to identify the user's input (i.e., **45** and **72**). As in prior programs, Lines 1–2 state the figure number, filename and purpose of the program.

```
1  // Fig. 2.7: Addition.java
2  // Addition program that inputs two numbers then displays their sum.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class Addition
6  {
7     // main method begins execution of Java application
8     public static void main(String[] args)
9     {
10       // create a Scanner to obtain input from the command window
11       Scanner input = new Scanner(System.in);
12
13       int number1; // first number to add
14       int number2; // second number to add
15       int sum; // sum of number1 and number2
16
17       System.out.print("Enter first integer: "); // prompt
18       number1 = input.nextInt(); // read first number from user
19
20       System.out.print("Enter second integer: "); // prompt
21       number2 = input.nextInt(); // read second number from user
22
23       sum = number1 + number2; // add numbers, then store total in sum
24
25       System.out.printf("Sum is %d%n", sum); // display sum
26    } // end method main
27 } // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.7 | Addition program that inputs two numbers then displays their sum.

2.5.1 `import` Declarations

A great strength of Java is its rich set of predefined classes that you can *reuse* rather than "reinventing the wheel." These classes are grouped into **packages**—*named groups of related classes*—and are collectively referred to as the **Java class library**, or the **Java Application Programming Interface** (**Java API**). Line 3

```
import java.util.Scanner; // program uses class Scanner
```

is an **`import` declaration** that helps the compiler locate a class that's used in this program. It indicates that this example uses Java's predefined `Scanner` class (discussed shortly) from the package that is named **`java.util`**. This enables the compiler to ensure that you use the class correctly.

Common Programming Error 2.7

All `import` declarations must appear before the first class declaration in the file. Placing an `import` declaration inside or after a class declaration is a syntax error.

Common Programming Error 2.8

Forgetting to include an `import` declaration for a class that must be imported results in a compilation error containing a message such as "`cannot find symbol`." When this occurs, check that you provided the proper `import` declarations and that the names in them are correct, including proper capitalization.

Software Engineering Observation 2.1

*In each new Java version, the APIs typically contain new capabilities that fix bugs, improve performance or offer better means for accomplishing tasks. The corresponding older versions are no longer needed and should not be used. Such APIs are said to be **deprecated** and might be removed from later Java versions.*

You'll often encounter deprecated APIs when browsing the online API documentation. The compiler will warn you when you compile code that uses deprecated APIs. If you compile your code with `javac` using the command-line argument `-deprecation`, the compiler will tell you which deprecated features you're using. For each one, the online documentation (`http://docs.oracle.com/javase/7/docs/api/`) indicates and typically links to the new feature that replaces the deprecated one.

2.5.2 Declaring Class `Addition`

Line 5

```
public class Addition
```

begins the declaration of class `Addition`. The filename for this `public` class must be `Addition.java`. Remember that the body of each class declaration starts with an opening left brace (line 6) and ends with a closing right brace (line 27).

The application begins execution with the `main` method (lines 8–26). The left brace (line 9) marks the beginning of method `main`'s body, and the corresponding right brace (line 26) marks its end. Method `main` is indented one level in the body of class `Addition`, and the code in the body of `main` is indented another level for readability.

2.5.3 Declaring and Creating a `Scanner` to Obtain User Input from the Keyboard

A **variable** is a location in the computer's memory where a value can be stored for use later in a program. All Java variables *must* be declared with a **name** and a **type** *before* they can be used. A variable's *name* enables the program to access the *value* of the variable in memory. A variable's name can be any valid identifier—again, a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that does *not* begin with a digit and does *not* contain spaces. A variable's *type* specifies what kind of information is stored at that location in memory. Like other statements, declaration statements end with a semicolon (;).

Line 11

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

is a **variable declaration statement** that specifies the *name* (`input`) and *type* (`Scanner`) of a variable that's used in this program. A **`Scanner`** enables a program to read data (e.g., numbers and strings) for use in a program. The data can come from many sources, such as the user at the keyboard or a file on disk. Before using a `Scanner`, you must create it and specify the *source* of the data.

The = in line 11 indicates that `Scanner` variable `input` should be **initialized** (i.e., prepared for use in the program) in its declaration with the result of the expression to the right of the equals sign—`new Scanner(System.in)`. This expression uses the **`new`** keyword to create a `Scanner` object that reads characters typed by the user at the keyboard. The **standard input object**, **`System.in`**, enables applications to read *bytes* of data typed by the user. The `Scanner` translates these bytes into types (like `int`s) that can be used in a program.

2.5.4 Declaring Variables to Store Integers

The variable declaration statements in lines 13–15

```
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` hold data of type **`int`**—they can hold *integer* values (whole numbers such as `72`, `-1127` and `0`). These variables are *not* yet initialized. The range of values for an `int` is –2,147,483,648 to +2,147,483,647. [*Note:* The `int` values you use in a program may not contain commas.]

Some other types of data are **`float`** and **`double`**, for holding real numbers, and **`char`**, for holding character data. Real numbers contain decimal points, such as in `3.4`, `0.0` and `-11.19`. Variables of type `char` represent individual characters, such as an uppercase letter (e.g., `A`), a digit (e.g., `7`), a special character (e.g., `*` or `%`) or an escape sequence (e.g., the tab character, `\t`). The types `int`, `float`, `double` and `char` are called **primitive types**. Primitive-type names are keywords and must appear in all lowercase letters. Appendix D summarizes the characteristics of the eight primitive types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`).

Several variables of the same type may be declared in a single declaration with the variable names separated by commas (i.e., a comma-separated list of variable names). For example, lines 13–15 can also be written as:

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

Good Programming Practice 2.7

Declare each variable in its own declaration. This format allows a descriptive comment to be inserted next to each variable being declared.

Good Programming Practice 2.8

Choosing meaningful variable names helps a program to be self-documenting (i.e., one can understand the program simply by reading it rather than by reading associated documentation or creating and viewing an excessive number of comments).

Good Programming Practice 2.9

By convention, variable-name identifiers begin with a lowercase letter, and every word in the name after the first word begins with a capital letter. For example, variable-name identifier `firstNumber` *starts its second word,* `Number`*, with a capital* `N`*. This naming convention is known as* ***camel case****, because the uppercase letters stand out like a camel's humps.*

2.5.5 Prompting the User for Input

Line 17

```
System.out.print("Enter first integer: "); // prompt
```

uses `System.out.print` to display the message `"Enter first integer: "`. This message is called a **prompt** because it directs the user to take a specific action. We use method `print` here rather than `println` so that the user's input appears on the same line as the prompt. Recall from Section 2.2 that identifiers starting with capital letters typically represent class names. Class `System` is part of package **`java.lang`**. Notice that class `System` is *not* imported with an `import` declaration at the beginning of the program.

Software Engineering Observation 2.2

By default, package `java.lang` *is imported in every Java program; thus, classes in* `java.lang` *are the only ones in the Java API that do not require an* `import` *declaration.*

2.5.6 Obtaining an `int` as Input from the User

Line 18

```
number1 = input.nextInt(); // read first number from user
```

uses `Scanner` object `input`'s `nextInt` method to obtain an integer from the user at the keyboard. At this point the program *waits* for the user to type the number and press the *Enter* key to submit the number to the program.

Our program assumes that the user enters a valid integer value. If not, a runtime logic error will occur and the program will terminate. Chapter 11, Exception Handling: A Deeper Look, 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*.

In line 18, we place the result of the call to method `nextInt` (an `int` value) in variable `number1` by using the **assignment operator**, `=`. The statement is read as "`number1` *gets* the value of `input.nextInt()`." Operator `=` is called a **binary operator**, because it has *two* **operands**—`number1` and the result of the method call `input.nextInt()`. This statement is called an *assignment statement*, because it *assigns* a value to a variable. Everything to the *right* of the assignment operator, `=`, is always evaluated *before* the assignment is performed.

Good Programming Practice 2.10

Place spaces on either side of a binary operator for readability.

2.5.7 Prompting for and Inputting a Second `int`

Line 20

```
System.out.print("Enter second integer: "); // prompt
```

prompts the user to enter the second integer. Line 21

```
number2 = input.nextInt(); // read second number from user
```

reads the second integer and assigns it to variable `number2`.

2.5.8 Using Variables in a Calculation

Line 23

```
sum = number1 + number2; // add numbers then store total in sum
```

is an assignment statement that calculates the sum of the variables `number1` and `number2` then assigns the result to variable `sum` by using the assignment operator, `=`. The statement is read as "`sum` *gets* the value of `number1 + number2`." When the program encounters the addition operation, it performs the calculation using the values stored in the variables `number1` and `number2`. In the preceding statement, the addition operator is a *binary operator*—its *two* operands are the variables `number1` and `number2`. Portions of statements that contain calculations are called **expressions**. In fact, an expression is any portion of a statement that has a *value* associated with it. For example, the value of the expression `number1 + number2` is the *sum* of the numbers. Similarly, the value of the expression `input.nextInt()` is the integer typed by the user.

2.5.9 Displaying the Result of the Calculation

After the calculation has been performed, line 25

```
System.out.printf("Sum is %d%n", sum); // display sum
```

uses method `System.out.printf` to display the `sum`. The format specifier **`%d`** is a *placeholder* for an `int` value (in this case the value of `sum`)—the letter `d` stands for "decimal integer." The remaining characters in the format string are all fixed text. So, method `printf` displays `"Sum is "`, followed by the value of `sum` (in the position of the `%d` format specifier) and a newline.

Calculations can also be performed *inside* `printf` statements. We could have combined the statements at lines 23 and 25 into the statement

```
System.out.printf("Sum is %d%n", (number1 + number2));
```

The parentheses around the expression `number1 + number2` are optional—they're included to emphasize that the value of the *entire* expression is output in the position of the `%d` format specifier. Such parentheses are said to be **redundant**.

2.5.10 Java API Documentation

For each new Java API class we use, we indicate the package in which it's located. This information 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

```
http://docs.oracle.com/javase/7/docs/api/index.html
```

You can download it from the Additional Resources section at

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

Appendix F shows how to use this documentation.

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** (in bytes) and a **value**.

In the addition program of Fig. 2.7, when the following statement (line 18) executes:

```
number1 = input.nextInt(); // read first number from user
```

the number typed by the user is placed into a memory location corresponding to the name `number1`. Suppose that the user enters `45`. The computer places that integer value into location `number1` (Fig. 2.8), replacing the previous value (if any) in that location. The previous value is lost, so this process is said to be *destructive*.

Fig. 2.8 | Memory location showing the name and value of variable `number1`.

When the statement (line 21)

```
number2 = input.nextInt(); // read second number from user
```

executes, suppose that the user enters `72`. The computer places that integer value into location `number2`. The memory now appears as shown in Fig. 2.9.

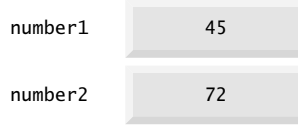


Fig. 2.9 | Memory locations after storing values for `number1` and `number2`.

After the program of Fig. 2.7 obtains values for `number1` and `number2`, it adds the values and places the total into variable `sum`. The statement (line 23)

```
sum = number1 + number2; // add numbers, then store total in sum
```

performs the addition, then replaces any previous value in `sum`. After `sum` has been calculated, memory appears as shown in Fig. 2.10. 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. When a value is read from a memory location, the process is *nondestructive*.

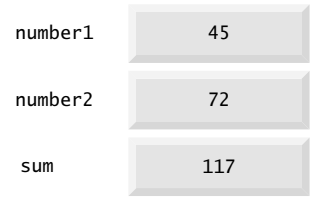


Fig. 2.10 | Memory locations after storing the sum of `number1` and `number2`.

2.7 Arithmetic

Most programs perform arithmetic calculations. The **arithmetic operators** are summarized in Fig. 2.11. Note the use of various special symbols not used in algebra. The **asterisk** (`*`) indicates multiplication, and the percent sign (`%`) is the **remainder operator**, which we'll discuss shortly. The arithmetic operators in Fig. 2.11 are *binary* operators, because each operates on *two* operands. For example, the expression `f + 7` contains the binary operator `+` and the two operands `f` and `7`.

Java operation	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`
Remainder	`%`	$r \bmod s$	`r % s`

Fig. 2.11 | Arithmetic operators.

Integer division yields an integer quotient. For example, the expression `7 / 4` evaluates to `1`, and the expression `17 / 5` evaluates to `3`. Any fractional part in integer division is simply *truncated* (i.e., *discarded*)—no *rounding* occurs. Java provides the remainder operator, `%`, which yields the remainder after 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 it can also be used with other arithmetic types. In this chapter's exercises and in later chapters, we consider several interesting applications of the remainder operator, such as determining whether one number is a multiple of another.

Arithmetic Expressions in Straight-Line Form

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 for Grouping Subexpressions

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

```
a * (b + c)
```

If an expression contains **nested parentheses**, such as

```
((a + b) * c)
```

the expression in the *innermost* set of parentheses (`a + b` in this case) is evaluated *first*.

Rules of Operator Precedence

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

1. Multiplication, division and remainder operations are applied first. If an expression contains several such operations, they're applied from left to right. Multiplication, division and remainder operators have the same level of precedence.
2. Addition and subtraction operations are applied next. If an expression contains several such operations, the operators are applied from left to right. Addition and subtraction operators have the same level of precedence.

These rules enable Java to apply operators in the correct *order*.[1] When we say that operators are applied from left to right, we're referring to their **associativity**. Some operators associate from right to left. Figure 2.12 summarizes these rules of operator precedence. A complete precedence chart is included in Appendix A.

Operator(s)	Operation(s)	Order of evaluation (precedence)
* / %	Multiplication Division Remainder	Evaluated first. If there are several operators of this type, they're evaluated from *left to right*.
+ -	Addition Subtraction	Evaluated next. If there are several operators of this type, they're evaluated from *left to right*.
=	Assignment	Evaluated last.

Fig. 2.12 | Precedence of arithmetic operators.

Sample Algebraic and Java Expressions

Now let's 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:

1. We use simple examples to explain the *order of evaluation* of expressions. Subtle issues occur in the more complex expressions you'll encounter later in the book. For more information on order of evaluation, see Chapter 15 of *The Java™ Language Specification* (`http://docs.oracle.com/javase/specs/jls/se7/html/index.html`).

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

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

The parentheses are required because division has higher precedence than 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

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

Here's 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 addition. The assignment occurs last because it has a lower precedence than multiplication or addition.

The following example contains remainder (%), 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 *, % and / operations are evaluated first in *left-to-right* order (i.e., they associate from left to right), because they have higher precedence than + and -. The + and - operations are evaluated next. These operations are also applied from *left to right*. The assignment (=) operation is evaluated last.

Evaluation of a Second-Degree Polynomial

To develop a better understanding of the rules of operator precedence, consider the evaluation of an assignment expression that includes a second-degree polynomial $ax^2 + bx + c$:

```
y = a * x * x + b * x + c;
    6   1   2   4   3   5
```

The multiplication operations are evaluated first in left-to-right order (i.e., they associate from left to right), because they have higher precedence than addition. (Java has no arithmetic operator for exponentiation, so x^2 is represented as `x * x`. Section 4.4 shows an alternative for performing exponentiation.) The addition operations are evaluated next from *left to right*. Suppose that `a`, `b`, `c` and `x` are initialized (given values) as follows: `a = 2`, `b = 3`, `c = 7` and `x = 5`. Figure 2.13 illustrates the order in which the operators are applied.

You can use *redundant parentheses* (unnecessary parentheses) to make an expression clearer. For example, the preceding statement might be parenthesized as follows:

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

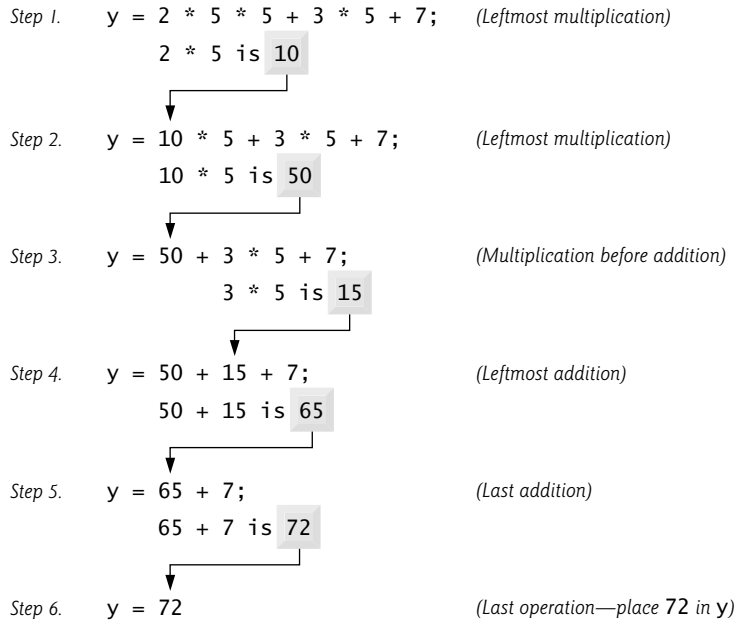


Fig. 2.13 | Order in which a second-degree polynomial is evaluated.

2.8 Decision Making: Equality and Relational Operators

A **condition** is an expression that can be **`true`** or **`false`**. This section introduces Java's **`if` selection statement**, which allows a program to make a **decision** based on a condition's value. For example, the condition "grade is greater than or equal to 60" determines whether a student passed a test. If the condition in an `if` statement is *true*, the body of the `if` statement executes. If the condition is *false*, the body does not execute. We'll see an example shortly.

Conditions in `if` statements can be formed by using the **equality operators** (`==` and `!=`) and **relational operators** (`>`, `<`, `>=` and `<=`) summarized in Fig. 2.14. Both equality operators have the same level of precedence, which is *lower* than that of the relational operators. The equality operators associate from *left to right*. The relational operators all have the same level of precedence and also associate from *left to right*.

Algebraic operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

Fig. 2.14 | Equality and relational operators. (Part 1 of 2.)

Algebraic operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Relational operators			
>	>	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. 2.14 | Equality and relational operators. (Part 2 of 2.)

Figure 2.15 uses six `if` statements to compare two integers input by the user. If the condition in any of these `if` statements is *true*, the statement associated with that `if` statement executes; otherwise, the statement is skipped. We use a `Scanner` to input the integers from the user and store them in variables `number1` and `number2`. The program *compares* the numbers and displays the results of the comparisons that are true.

```
1  // Fig. 2.15: Comparison.java
2  // Compare integers using if statements, relational operators
3  // and equality operators.
4  import java.util.Scanner; // program uses class Scanner
5
6  public class Comparison
7  {
8     // main method begins execution of Java application
9     public static void main(String[] args)
10    {
11       // create Scanner to obtain input from command line
12       Scanner input = new Scanner(System.in);
13
14       int number1; // first number to compare
15       int number2; // second number to compare
16
17       System.out.print("Enter first integer: "); // prompt
18       number1 = input.nextInt(); // read first number from user
19
20       System.out.print("Enter second integer: "); // prompt
21       number2 = input.nextInt(); // read second number from user
22
23       if (number1 == number2)
24          System.out.printf("%d == %d%n", number1, number2);
25
26       if (number1 != number2)
27          System.out.printf("%d != %d%n", number1, number2);
28
29       if (number1 < number2)
30          System.out.printf("%d < %d%n", number1, number2);
```

Fig. 2.15 | Compare integers using `if` statements, relational operators and equality operators. (Part 1 of 2.)

```
31
32          if (number1 > number2)
33             System.out.printf("%d > %d%n", number1, number2);
34
35          if (number1 <= number2)
36             System.out.printf("%d <= %d%n", number1, number2);
37
38          if (number1 >= number2)
39             System.out.printf("%d >= %d%n", number1, number2);
40       } // end method main
41    } // end class Comparison
```

```
Enter first integer: 777
Enter second integer: 777
777 == 777
777 <= 777
777 >= 777
```

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Fig. 2.15 | Compare integers using `if` statements, relational operators and equality operators. (Part 2 of 2.)

The declaration of class `Comparison` begins at line 6

```
public class Comparison
```

The class's `main` method (lines 9–40) begins the execution of the program. Line 12

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

declares `Scanner` variable `input` and assigns it a `Scanner` that inputs data from the standard input (i.e., the keyboard).

Lines 14–15

```
int number1; // first number to compare
int number2; // second number to compare
```

declare the `int` variables used to store the values input from the user.

Lines 17–18

```
System.out.print("Enter first integer: "); // prompt
number1 = input.nextInt(); // read first number from user
```

prompt the user to enter the first integer and input the value, respectively. The value is stored in variable `number1`.

Lines 20–21

```
System.out.print("Enter second integer: "); // prompt
number2 = input.nextInt(); // read second number from user
```

prompt the user to enter the second integer and input the value, respectively. The value is stored in variable `number2`.

Lines 23–24

```
if (number1 == number2)
   System.out.printf("%d == %d%n", number1, number2);
```

compare the values of `number1` and `number2` to determine whether they're equal. An `if` statement always begins with keyword `if`, followed by a condition in parentheses. An `if` statement expects one statement in its body, but may contain multiple statements if they're enclosed in a set of braces (`{}`). The indentation of the body statement shown here is not required, but it improves the program's readability by emphasizing that the statement in line 24 *is part of* the `if` statement that begins at line 23. Line 24 executes only if the numbers stored in variables `number1` and `number2` are equal (i.e., the condition is *true*). The `if` statements in lines 26–27, 29–30, 32–33, 35–36 and 38–39 compare `number1` and `number2` using the operators `!=`, `<`, `>`, `<=` and `>=`, respectively. If the condition in one or more of the `if` statements is *true*, the corresponding body statement executes.

Common Programming Error 2.9

Confusing the equality operator, ==, with the assignment operator, =, can cause a logic error or a compilation error. The equality operator should be read as "is equal to" and the assignment operator as "gets" or "gets the value of." To avoid confusion, some people read the equality operator as "double equals" or "equals equals."

Good Programming Practice 2.11

Place only one statement per line in a program for readability.

There's no semicolon (`;`) at the end of the first line of each `if` statement. Such a semicolon would result in a logic error at execution time. For example,

```
if (number1 == number2); // logic error
   System.out.printf("%d == %d%n", number1, number2);
```

would actually be interpreted by Java as

```
if (number1 == number2)
   ; // empty statement
System.out.printf("%d == %d%n", 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` statement is *true*. When the empty statement executes, no task is performed. The program then continues with the output statement, which *always* executes, *regardless* of whether the condition is true or false, because the output statement is *not* part of the `if` statement.

White space

Note the use of white space in Fig. 2.15. Recall that the compiler normally ignores white space. So, statements may be split over several lines and may be spaced according to your preferences without affecting a program's meaning. It's incorrect to split identifiers and strings. Ideally, statements should be kept small, but this is not always possible.

Error-Prevention Tip 2.4

A lengthy statement can be spread over several lines. If a single statement must be split across lines, choose natural breaking points, 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.

Operators Discussed So Far

Figure 2.16 shows the operators discussed so far in decreasing order of precedence. All but the assignment operator, `=`, associate from *left to right*. The assignment operator, `=`, associates from *right to left*. An assignment expression's value is whatever was assigned to the variable on the `=` operator's left side—for example, the value of the expression `x = 7` is `7`. So an expression like `x = y = 0` is evaluated as if it had been written as `x = (y = 0)`, which first assigns the value `0` to variable `y`, then assigns the result of that assignment, `0`, to `x`.

Good Programming Practice 2.12

When writing expressions containing many operators, refer to the operator precedence chart (Appendix A). Confirm that the operations in the expression are performed in the order you expect. If, in a complex expression, you're uncertain about the order of evaluation, use parentheses to force the order, exactly as you'd do in algebraic expressions.

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

Fig. 2.16 | Precedence and associativity of operators discussed.

2.9 (Optional) GUI and Graphics Case Study: Using Dialog Boxes

This optional case study is designed for those who want to begin learning Java's capabilities for creating graphical user interfaces (GUIs) and graphics early in the book, before the deeper discussions of these topics later in the book. This case study features Java's mature Swing technology, which as of this writing is still a bit more popular than the newer JavaFX technology presented in later chapters.

This GUI and Graphics Case Study appears in 10 brief sections (see Fig. 2.17). Each section introduces new concepts and provides examples with screen captures that show

sample interactions and results. In the first few sections, you'll create your first graphical apps. In subsequent sections, you'll use object-oriented programming concepts to create an app that draws a variety of shapes. When we formally introduce GUIs in Chapter 12, we use the mouse to choose exactly which shapes to draw and where to draw them. In Chapter 13, we add capabilities of the Java 2D graphics API to draw the shapes with different line thicknesses and fills. We hope you find this case study informative and entertaining.

Location	Title—Exercise(s)
Section 2.4	Using Dialog Boxes—Basic input and output with dialog boxes
Section 3.14	Creating Simple Drawings—Displaying and drawing lines on the screen
Section 4.10	Drawing Rectangles and Ovals—Using shapes to represent data
Section 5.13	Colors and Filled Shapes—Drawing a bull's-eye and random graphics
Section 6.14	Drawing Arcs—Drawing spirals with arcs
Section 8.16	Using Objects with Graphics—Storing shapes as objects
Section 9.7	Displaying Text and Images Using Labels—Providing status information
Section 10.11	Drawing with Polymorphism—Identifying the similarities between shapes
Exercise 12.17	Expanding the Interface—Using GUI components and event handling
Exercise 13.31	Adding Java 2D—Using the Java 2D API to enhance drawings

Fig. 2.17 | Summary of the GUI and Graphics Case Study in each chapter.

Displaying Text in a Dialog Box

The programs presented thus far display output in the *command window*. Many apps use windows or **dialog boxes** (also called **dialogs**) to display output. Web browsers such as Chrome, Firefox, Internet Explorer, Safari and Opera display web pages in their own windows. E-mail programs allow you to type and read messages in a window. Typically, dialog boxes are windows in which programs display important messages to users. Class **`JOptionPane`** provides prebuilt dialog boxes that enable programs to display windows containing messages—such windows are called **message dialogs**. Figure 2.18 displays the `String` `"Welcome to Java"` in a message dialog.

```
1  // Fig. 2.18: Dialog1.java
2  // Using JOptionPane to display multiple lines in a dialog box.
3  import javax.swing.JOptionPane;
4
5  public class Dialog1
6  {
7     public static void main(String[] args)
8     {
9        // display a dialog with a message
10       JOptionPane.showMessageDialog(null, "Welcome to Java");
11    }
12 } // end class Dialog1
```

Fig. 2.18 | Using `JOptionPane` to display multiple lines in a dialog box. (Part 1 of 2.)

Fig. 2.18 | Using `JOptionPane` to display multiple lines in a dialog box. (Part 2 of 2.)

`JOptionPane` Class `static` Method `showMessagDialog`

Line 3 indicates that the program uses class `JOptionPane` from package **`javax.swing`**. This package contains many classes that help you create **graphical user interfaces** (**GUIs**). **GUI components** facilitate data entry by a program's user and presentation of outputs to the user. Line 10 calls `JOptionPane` method **`showMessageDialog`** to display a dialog box containing a message. The method requires two arguments. The first helps the Java app determine where to position the dialog box. A dialog is typically displayed from a GUI app with its own window. The first argument refers to that window (known as the *parent window*) and causes the dialog to appear centered over the app's window. If the first argument is `null`, the dialog box is displayed at the center of your screen. The second argument is the `String` to display in the dialog box.

Introducing `static` Methods

`JOptionPane` method `showMessageDialog` is a so-called **`static` method**. Such methods often define frequently used tasks. For example, many programs display dialog boxes, and the code to do this is the same each time. Rather than requiring you to "reinvent the wheel" and create code to display a dialog, the designers of class `JOptionPane` declared a `static` method that performs this task for you. A `static` method is called by using its class name followed by a dot (`.`) and the method name, as in

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

Notice that you do *not* create an object of class `JOptionPane` to use its `static` method `showMessageDialog`. We discuss `static` methods in more detail in Chapter 5.

Entering Text in a Dialog

Figure 2.19 uses another predefined `JOptionPane` dialog called an **input dialog** that allows the user to *enter data* into a program. The program asks for the user's name and responds with a message dialog containing a greeting and the name that the user entered.

```
1  // Fig. 2.19: NameDialog.java
2  // Obtaining user input from a dialog.
3  import javax.swing.JOptionPane;
4
5  public class NameDialog
6  {
7     public static void main(String[] args)
8     {
```

Fig. 2.19 | Obtaining user input from a dialog. (Part 1 of 2.)

```
 9         // prompt user to enter name
10         String name = JOptionPane.showInputDialog("What is your name?");
11
12         // create the message
13         String message =
14            String.format("Welcome, %s, to Java Programming!", name);
15
16         // display the message to welcome the user by name
17         JOptionPane.showMessageDialog(null, message);
18      } // end main
19   } // end class NameDialog
```

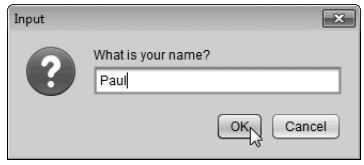


Fig. 2.19 | Obtaining user input from a dialog. (Part 2 of 2.)

`JOptionPane` Class `static` Method `showInputDialog`

Line 10 uses `JOptionPane` method **`showInputDialog`** to display an input dialog containing a prompt and a *field* (known as a **text field**) in which you can enter text. Method `showInputDialog`'s argument is the prompt that indicates what you should enter. You type characters in the text field, then click the **OK** button or press the *Enter* key to return the `String` to the program. Method `showInputDialog` returns a `String` containing the characters you typed. We store the `String` in variable `name`. If you press the dialog's **Cancel** button or press the *Esc* key, the method returns `null` and the program displays the word "null" as the name.

`String` Class `static` Method `format`

Lines 13–14 use `static` `String` method **`format`** to return a `String` containing a greeting with the user's name. Method `format` works like method `System.out.printf`, except that `format` returns the formatted `String` rather than displaying it in a command window. Line 17 displays the greeting in a message dialog, just as we did in Fig. 2.18.

GUI and Graphics Case Study Exercise

2.1 Modify the addition program in Fig. 2.7 to use dialog-based input and output with the methods of class `JOptionPane`. Since method `showInputDialog` returns a `String`, you must convert the `String` the user enters to an `int` for use in calculations. The `static` method `parseInt` of class `Integer` (package `java.lang`) takes a `String` argument representing an integer and returns the value as an `int`. If the `String` does not contain a valid integer, the program will terminate with an error.

2.10 Wrap-Up

In this chapter, you learned many important features of Java, including displaying data on the screen in a **Command Prompt**, inputting data from the keyboard, performing calculations and making decisions. The applications presented here introduced you to many basic programming concepts. In the next chapter we begin our introduction to control statements, which specify the order in which a program's actions are performed.

Summary

Section 2.2 Your First Program in Java: Printing a Line of Text

- A Java application (p. 35) executes when you use the `java` command to launch the JVM.
- Comments (p. 36) document programs and improve their readability. The compiler ignores them.
- A comment that begins with `//` is an end-of-line comment—it terminates at the end of the line on which it appears.
- Traditional comments (p. 36) can be spread over several lines and are delimited by `/*` and `*/`.
- Javadoc comments (p. 36), delimited by `/**` and `*/`, enable you to embed program documentation in your code. The `javadoc` utility program generates HTML pages based on these comments.
- A syntax error (p. 36; also called a compiler error, compile-time error or compilation error) occurs when the compiler encounters code that violates Java's language rules. It's similar to a grammar error in a natural language.
- Blank lines, space characters and tab characters are known as white space (p. 37). White space makes programs easier to read and is ignored by the compiler.
- Keywords (p. 37) are reserved for use by Java and are always spelled with all lowercase letters.
- Keyword `class` (p. 37) introduces a class declaration.
- By convention, all class names in Java begin with a capital letter and capitalize the first letter of each word they include (e.g., `SampleClassName`).
- A Java class name is an identifier—a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that does not begin with a digit and does not contain spaces.
- Java is case sensitive (p. 37)—that is, uppercase and lowercase letters are distinct.
- The body of every class declaration (p. 37) is delimited by braces, `{` and `}`.
- A `public` (p. 37) class declaration must be saved in a file with the same name as the class followed by the "`.java`" file-name extension.
- Method `main` (p. 38) is the starting point of every Java application and must begin with

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

  otherwise, the JVM will not execute the application.
- Methods perform tasks and return information when they complete them. Keyword `void` (p. 38) indicates that a method will perform a task but return no information.
- Statements instruct the computer to perform actions.
- A string (p. 39) in double quotes is sometimes called a character string or a string literal.
- The standard output object (`System.out`; p. 39) displays characters in the command window.
- Method `System.out.println` (p. 39) displays its argument (p. 39) in the command window followed by a newline character to position the output cursor to the beginning of the next line.
- You compile a program with the command `javac`. If the program contains no syntax errors, a class file (p. 40) containing the Java bytecodes that represent the application is created. These bytecodes are interpreted by the JVM when you execute the program.
- To run an application, type `java` followed by the name of the class that contains the `main` method.

Section 2.3 Modifying Your First Java Program

- `System.out.print` (p. 42) displays its argument and positions the output cursor immediately after the last character displayed.
- A backslash (\) in a string is an escape character (p. 43). Java combines it with the next character to form an escape sequence (p. 43). The escape sequence `\n` (p. 43) represents the newline character.

Section 2.4 Displaying Text with `printf`

- `System.out.printf` method (p. 43; `f` means "formatted") displays formatted data.
- Method `printf`'s first argument is a format string (p. 44) containing fixed text and/or format specifiers. Each format specifier (p. 44) indicates the type of data to output and is a placeholder for a corresponding argument that appears after the format string.
- Format specifiers begin with a percent sign (`%`) and are followed by a character that represents the data type. The format specifier `%s` (p. 44) is a placeholder for a string.
- The `%n` format specifier (p. 44) is a portable line separator. You cannot use `%n` in the argument to `System.out.print` or `System.out.println`; however, the line separator output by `System.out.println` after it displays its argument is portable across operating systems.

Section 2.5 Another Application: Adding Integers

- An `import` declaration (p. 45) helps the compiler locate a class that's used in a program.
- Java's rich set of predefined classes are grouped into packages (p. 45)—named groups of classes. These are referred to as the Java class library (p. 45), or the Java Application Programming Interface (Java API).
- A variable (p. 46) is a location in the computer's memory where a value can be stored for use later in a program. All variables must be declared with a name and a type before they can be used.
- A variable's name enables the program to access the variable's value in memory.
- A `Scanner` (package `java.util`; p. 47) enables a program to read data that the program will use. Before a `Scanner` can be used, the program must create it and specify the source of the data.
- Variables should be initialized (p. 47) to prepare them for use in a program.
- The expression `new Scanner(System.in)` creates a `Scanner` that reads from the standard input object (`System.in`; p. 47)—normally the keyboard.
- Data type `int` (p. 47) is used to declare variables that will hold integer values. The range of values for an `int` is –2,147,483,648 to +2,147,483,647.
- Types `float` and `double` (p. 47) specify real numbers with decimal points, such as `3.4` and `-11.19`.
- Variables of type `char` (p. 47) represent individual characters, such as an uppercase letter (e.g., `A`), a digit (e.g., `7`), a special character (e.g., `*` or `%`) or an escape sequence (e.g., tab, `\t`).
- Types such as `int`, `float`, `double` and `char` are primitive types (p. 47). Primitive-type names are keywords; thus, they must appear in all lowercase letters.
- A prompt (p. 48) directs the user to take a specific action.
- `Scanner` method `nextInt` obtains an integer for use in a program.
- The assignment operator, `=` (p. 48), enables the program to give a value to a variable. It's called a binary operator (p. 48) because it has two operands.
- Portions of statements that have values are called expressions (p. 49).
- The format specifier `%d` (p. 49) is a placeholder for an `int` value.

Section 2.6 Memory Concepts

- Variable names (p. 50) correspond to locations in the computer's memory. Every variable has a name, a type, a size and a value.
- A value that's placed in a memory location replaces the location's previous value, which is lost.

Section 2.7 Arithmetic

- The arithmetic operators (p. 51) are `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division) and `%` (remainder).

- Integer division (p. 51) yields an integer quotient.
- The remainder operator, `%` (p. 51), yields the remainder after division.
- Arithmetic expressions must be written in straight-line form (p. 51).
- If an expression contains nested parentheses (p. 52), the innermost set is evaluated first.
- Java applies the operators in arithmetic expressions in a precise sequence determined by the rules of operator precedence (p. 52).
- When we say that operators are applied from left to right, we're referring to their associativity (p. 52). Some operators associate from right to left.
- Redundant parentheses (p. 53) can make an expression clearer.

Section 2.8 Decision Making: Equality and Relational Operators

- The `if` statement (p. 54) makes a decision based on a condition's value (true or false).
- Conditions in `if` statements can be formed by using the equality (`==` and `!=`) and relational (`>`, `<`, `>=` and `<=`) operators (p. 54).
- An `if` statement begins with keyword `if` followed by a condition in parentheses and expects one statement in its body.
- The empty statement (p. 57) is a statement that does not perform a task.

Self-Review Exercises

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

a) A(n) _____ begins the body of every method, and a(n) _____ ends the body of every method.
b) You can use the _____ statement to make decisions.
c) _____ begins an end-of-line comment.
d) _____, _____ and _____ are called white space.
e) _____ are reserved for use by Java.
f) Java applications begin execution at method _____.
g) Methods _____, _____ and _____ display information in a command window.

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

a) Comments cause the computer to print the text after the `//` on the screen when the program executes.
b) All variables must be given a type when they're declared.
c) Java considers the variables `number` and `NuMbEr` to be identical.
d) The remainder operator (`%`) can be used only with integer operands.
e) The arithmetic operators `*`, `/`, `%`, `+` and `-` all have the same level of precedence.

2.3 Write statements to accomplish each of the following tasks:

a) Declare variables `c`, `thisIsAVariable`, `q76354` and `number` to be of type `int`.
b) Prompt the user to enter an integer.
c) Input an integer and assign the result to `int` variable `value`. Assume `Scanner` variable `input` can be used to read a value from the keyboard.
d) Print `"This is a Java program"` on one line in the command window. Use method `System.out.println`.
e) Print `"This is a Java program"` on two lines in the command window. The first line should end with `Java`. Use method `System.out.printf` and two `%s` format specifiers.
f) If the variable `number` is not equal to `7`, display `"The variable number is not equal to 7"`.

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

a)
```
if (c < 7);
   System.out.println("c is less than 7");
```

b)
```
if (c => 7)
   System.out.println("c is equal to or greater than 7");
```

2.5 Write declarations, statements or comments that accomplish each of the following tasks:

a) State that a program will calculate the product of three integers.
b) Create a `Scanner` called `input` that reads values from the standard input.
c) Declare the variables `x`, `y`, `z` and `result` to be of type `int`.
d) Prompt the user to enter the first integer.
e) Read the first integer from the user and store it in the variable `x`.
f) Prompt the user to enter the second integer.
g) Read the second integer from the user and store it in the variable `y`.
h) Prompt the user to enter the third integer.
i) Read the third integer from the user and store it in the variable `z`.
j) Compute the product of the three integers contained in variables `x`, `y` and `z`, and assign the result to the variable `result`.
k) Use `System.out.printf` to display the message `"Product is"` followed by the value of the variable `result`.

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) `if`. c) `//`. d) Space characters, newlines and tabs. e) Keywords. f) `main`. g) `System.out.print`, `System.out.println` and `System.out.printf`.

2.2 a) False. Comments do not cause any action to be performed when the program executes. They're used to document programs and improve their readability.
b) True.
c) False. Java is case sensitive, so these variables are distinct.
d) False. The remainder operator can also be used with noninteger operands in Java.
e) False. The operators `*`, `/` and `%` are higher precedence than operators `+` and `-`.

2.3 a)
```
int c, thisIsAVariable, q76354, number;
```
or
```
int c;
int thisIsAVariable;
int q76354;
int number;
```

b)
```
System.out.print("Enter an integer: ");
```

c)
```
value = input.nextInt();
```

d)
```
System.out.println("This is a Java program");
```

e)
```
System.out.printf("%s%n%s%n", "This is a Java", "program");
```

f)
```
if (number != 7)
   System.out.println("The variable number is not equal to 7");
```

2.4 a) Error: Semicolon after the right parenthesis of the condition `(c < 7)` in the `if`.
Correction: Remove the semicolon after the right parenthesis. [*Note:* As a result, the output statement will execute regardless of whether the condition in the `if` is true.]
b) Error: The relational operator `=>` is incorrect. Correction: Change `=>` to `>=`.

2.5 a)
```
// Calculate the product of three integers
```
b)
```
Scanner input = new Scanner(System.in);
```
c)
```
int x, y, z, result;
```
or
```
int x;
int y;
int z;
int result;
```
d)
```
System.out.print("Enter first integer: ");
```
e)
```
x = input.nextInt();
```
f)
```
System.out.print("Enter second integer: ");
```
g)
```
y = input.nextInt();
```
h)
```
System.out.print("Enter third integer: ");
```
i)
```
z = input.nextInt();
```
j)
```
result = x * y * z;
```
k)
```
System.out.printf("Product is %d%n", result);
```

2.6 The solution to Self-Review Exercise 2.6 is as follows:

```
1  // Ex. 2.6: Product.java
2  // Calculate the product of three integers.
3  import java.util.Scanner; // program uses Scanner
4
5  public class Product
6  {
7     public static void main(String[] args)
8     {
9        // create Scanner to obtain input from command window
10       Scanner input = new Scanner(System.in);
11
12       int x; // first number input by user
13       int y; // second number input by user
14       int z; // third number input by user
15       int result; // product of numbers
16
17       System.out.print("Enter first integer: "); // prompt for input
18       x = input.nextInt(); // read first integer
19
20       System.out.print("Enter second integer: "); // prompt for input
21       y = input.nextInt(); // read second integer
22
23       System.out.print("Enter third integer: "); // prompt for input
24       z = input.nextInt(); // read third integer
25
26       result = x * y * z; // calculate product of numbers
27
28       System.out.printf("Product is %d%n", result);
29    } // end method main
30 } // end class Product
```

```
Enter first integer: 10
Enter second integer: 20
Enter third integer: 30
Product is 6000
```

Exercises

2.7 Fill in the blanks in each of the following statements:
a) _____ are used to document a program and improve its readability.
b) A decision can be made in a Java program with a(n) _____.
c) Calculations are normally performed by _____ statements.
d) The arithmetic operators with the same precedence as multiplication are _____ and _____.
e) When parentheses in an arithmetic expression are nested, the _____ set of parentheses is evaluated first.
f) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a(n) _____.

2.8 Write Java statements that accomplish each of the following tasks:
a) Display the message `"Enter an integer: "`, leaving the cursor on the same line.
b) Assign the product of variables `b` and `c` to variable `a`.
c) Use a comment to state that a program performs a sample payroll calculation.

2.9 State whether each of the following is *true* or *false*. If *false*, explain why.
a) Java operators are evaluated from left to right.
b) The following are all valid variable names: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales$`, `his_$account_total`, `a`, `b$`, `c`, `z` and `z2`.
c) A valid Java arithmetic expression with no parentheses is evaluated from left to right.
d) The following are all invalid variable names: `3g`, `87`, `67h2`, `h22` and `2h`.

2.10 Assuming that `x = 2` and `y = 3`, what does each of the following statements display?
a) `System.out.printf("x = %d%n", x);`
b) `System.out.printf("Value of %d + %d is %d%n", x, x, (x + x));`
c) `System.out.printf("x =");`
d) `System.out.printf("%d = %d%n", (x + y), (y + x));`

2.11 Which of the following Java statements contain variables whose values are modified?
a) `p = i + j + k + 7;`
b) `System.out.println("variables whose values are modified");`
c) `System.out.println("a = 5");`
d) `value = input.nextInt();`

2.12 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);`

2.13 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))));`

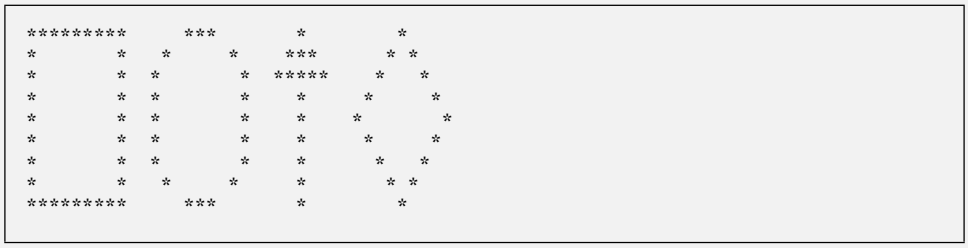
2.14 Write an application that displays the numbers 1 to 4 on the same line, with each pair of adjacent numbers separated by one space. Use the following techniques:
a) Use one `System.out.println` statement.
b) Use four `System.out.print` statements.
c) Use one `System.out.printf` statement.

2.15 *(Arithmetic)* Write an application that asks the user to enter two integers, obtains them from the user and prints their sum, product, difference and quotient (division). Use the techniques shown in Fig. 2.7.

2.16 *(Comparing Integers)* Write an application that asks the user to enter two integers, obtains them from the user and displays the larger number followed by the words "`is larger`". If the numbers are equal, print the message "`These numbers are equal`". Use the techniques shown in Fig. 2.15.

2.17 *(Arithmetic, Smallest and Largest)* Write an application that inputs three integers from the user and displays the sum, average, product, smallest and largest of the numbers. Use the techniques shown in Fig. 2.15. [*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….]

2.18 *(Displaying Shapes with Asterisks)* Write an application that displays a box, an oval, an arrow and a diamond using asterisks (*), as follows:



2.19 What does the following code print?

```
System.out.printf("*%n**%n***%n****%n*****%n");
```

2.20 What does the following code print?

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

2.21 What does the following code print?

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

2.22 What does the following code print?

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

2.23 What does the following code print?

```
System.out.printf("%s%n%s%n%s%n", "*", "***", "*****");
```

2.24 *(Largest and Smallest Integers)* Write an application that reads five integers and determines and prints the largest and smallest integers in the group. Use only the programming techniques you learned in this chapter.

2.25 *(Odd or Even)* Write an application that reads an integer and determines and prints whether it's odd or even. [*Hint:* Use the remainder operator. An even number is a multiple of 2. Any multiple of 2 leaves a remainder of 0 when divided by 2.]

2.26 *(Multiples)* Write an application that reads two integers, determines whether the first is a multiple of the second and prints the result. [*Hint:* Use the remainder operator.]

2.27 *(Checkerboard Pattern of Asterisks)* Write an application that displays a checkerboard pattern, as follows:

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

2.28 *(Diameter, Circumference and Area of a Circle)* Here's a peek ahead. In this chapter, you learned about integers and the type `int`. Java can also represent floating-point numbers that contain decimal points, such as 3.14159. Write an application that inputs from the user the radius of a circle as an integer and prints the circle's diameter, circumference and area using the floating-point value 3.14159 for π. Use the techniques shown in Fig. 2.7. [*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 package `java.lang`. Classes in that package are imported automatically, so you do not need to `import` class `Math` to use it.] Use the following formulas (r is the radius):

$diameter = 2r$
$circumference = 2\pi r$
$area = \pi r^2$

Do not store the results of each calculation in a variable. Rather, specify each calculation as the value that will be output in a `System.out.printf` statement. The values produced by the circumference and area calculations are floating-point numbers. Such values can be output with the format specifier `%f` in a `System.out.printf` statement. You'll learn more about floating-point numbers in Chapter 3.

2.29 *(Integer Value of a Character)* Here's another peek ahead. In this chapter, you learned about integers and the 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 together with 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 a character's integer equivalent by preceding that character with `(int)`, as in

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

An operator of this form is called a cast operator. (You'll learn about cast operators in Chapter 3.) The following statement outputs a character and its integer equivalent:

```
System.out.printf("The character %c has the value %d%n", 'A', ((int) 'A'));
```

When the preceding statement executes, it displays the character `A` and the value `65` (from the Unicode® character set) as part of the string. The format specifier `%c` is a placeholder for a character (in this case, the character `'A'`).

Using statements similar to the one shown earlier in this exercise, write an application that displays the integer equivalents of some uppercase letters, lowercase letters, digits and special symbols. Display the integer equivalents of the following: `A B C a b c 0 1 2 $ * + /` and the blank character.

2.30 ***(Separating the Digits in an Integer)*** 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
```

Assume that the user enters the correct number of digits. What happens when you enter a number with more than five digits? What happens when you enter a number with fewer than five digits? [*Hint:* It's possible to do this exercise with the techniques you learned in this chapter. You'll need to use both division and remainder operations to "pick off" each digit.]

2.31 ***(Table of Squares and Cubes)*** 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 shown below.

```
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
```

2.32 ***(Negative, Positive and Zero Values)*** 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.

Making a Difference

2.33 ***(Body Mass Index Calculator)*** We introduced the body mass index (BMI) calculator in Exercise 1.10. The formulas for calculating BMI are

$$BMI = \frac{weightInPounds \times 703}{heightInInches \times heightInInches}$$

or

$$BMI = \frac{weightInKilograms}{heightInMeters \times heightInMeters}$$

Create a BMI calculator that reads the user's weight in pounds and height in inches (or, if you prefer, the user's weight in kilograms and height in meters), then calculates and displays the user's body mass index. Also, display the following information from the Department of Health and Human Services/National Institutes of Health so the user can evaluate his/her BMI:

```
BMI VALUES
Underweight: less than 18.5
Normal:      between 18.5 and 24.9
Overweight:  between 25 and 29.9
Obese:       30 or greater
```

[*Note:* In this chapter, you learned to use the `int` type to represent whole numbers. The BMI calculations when done with `int` values will both produce whole-number results. In Chapter 3, you'll learn to use the `double` type to represent numbers with decimal points. When the BMI calculations are performed with `double`s, they'll both produce numbers with decimal points—these are called "floating-point" numbers.]

2.34 ***(World Population Growth Calculator)*** Use the web to determine the current world population and the annual world population growth rate. Write an application that inputs these values, then displays the estimated world population after one, two, three, four and five years.

2.35 ***(Car-Pool Savings Calculator)*** Research several car-pooling websites. Create an application that calculates your daily driving cost, so that you can estimate how much money could be saved by car pooling, which also has other advantages such as reducing carbon emissions and reducing traffic congestion. The application should input the following information and display the user's cost per day of driving to work:

a) Total miles driven per day.
b) Cost per gallon of gasoline.
c) Average miles per gallon.
d) Parking fees per day.
e) Tolls per day.