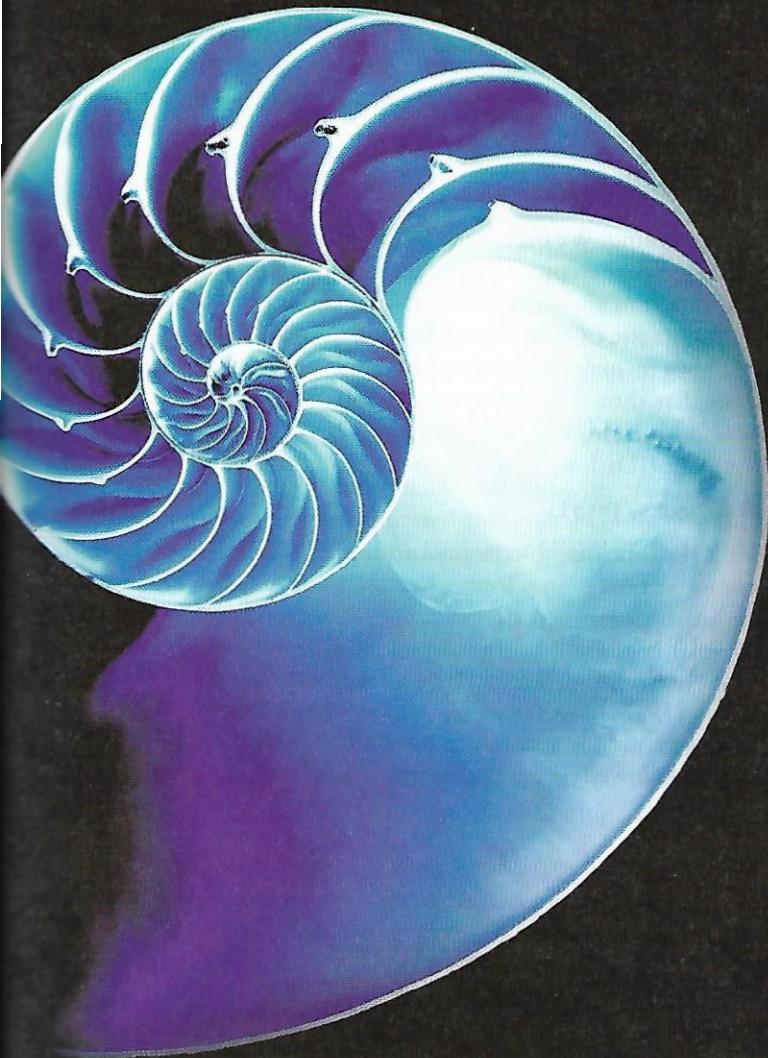


# Introduction to Java Applications; Input/Output and Operators



# 2

## 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
  - 2.2** Your First Program in Java: Printing a Line of Text
    - 2.2.1 Compiling the Application
    - 2.2.2 Executing the Application
  - 2.3** Modifying Your First Java Program
  - 2.4** Displaying Text with `printf`
  - 2.5** Another Application: Adding Integers
    - 2.5.1 `import` Declarations
    - 2.5.2 Declaring and Creating a `Scanner` to Obtain User Input from the Keyboard
    - 2.5.3 Prompting the User for Input
  - 2.5.4 Declaring a Variable to Store an Integer and Obtaining an Integer from the Keyboard
  - 2.5.5 Obtaining a Second Integer
  - 2.5.6 Using Variables in a Calculation
  - 2.5.7 Displaying the Calculation Result
  - 2.5.8 Java API Documentation
  - 2.5.9 Declaring and Initializing Variables in Separate Statements
  - 2.6** Memory Concepts
  - 2.7** Arithmetic
  - 2.8** Decision Making: Equality and Relational Operators
  - 2.9** Wrap-Up

[Summary](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#) | [Making a Difference](#)

---

## 2.1 Introduction

This chapter introduces Java 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 getting-started videos at

<http://www.deitel.com/books/jhtp11>

for the three most popular Java IDEs—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). Sections 2.2.1–2.2.2 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     // main method begins execution of Java application
6     public static void main(String[] args) {
7         System.out.println("Welcome to Java Programming!");
8     } // end method main
9 } // end class Welcome1

```

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

```
Welcome to Java Programming!
```

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

We use line numbers for instructional purposes—they’re *not* part of a Java program. This example illustrates several important Java features. We’ll see that line 7 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 the program’s 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 5, 8 and 9). 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 with the delimiter /\* and end with \*/. 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.

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 HTML5 web-page format. We use // comments throughout our code, rather than traditional or Javadoc comments, to save space. 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 natural-language grammar rules specifying sentence structure, such as those in English, French, Spanish, etc. 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

Blank lines (like line 3), space characters and tabs can make programs easier to read. Together, they're known as white space. The compiler ignores white space.



### Good Programming Practice 2.2

*Use white space 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 define. The `class` keyword introduces a class declaration and is immediately followed by the class name (`Welcome1`). Keywords are reserved for use by Java and are spelled with all lowercase letters. The complete list of keywords is shown in Appendix C.

In Chapters 2–7, every class we define begins with the `public` keyword. For now, we simply require it. 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 the 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.



### Good Programming Practice 2.3

*By convention, every word in a class-name identifier begins with an uppercase letter. For example, the class-name identifier `DollarAmount` starts its first word, `Dollar`, with an uppercase D and its second word, `Amount`, with an uppercase A. This naming convention is known as camel case, because the uppercase letters stand out like a camel's humps.*

### *Underscore (\_) in Java 9*

As of Java 9, you can no longer use an underscore (\_) by itself as an identifier.

9

#### *Class Body*

A left brace (at the end of line 4), {, begins the body of every class declaration. A corresponding right brace (at line 9), }, must end each class declaration. Lines 5–8 are indented.



#### **Good Programming Practice 2.4**

*Indent the entire body of each class declaration one “level” between the braces that delimit the class’s. 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.*



#### **Good Programming Practice 2.5**

*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.*



#### **Common Programming Error 2.3**

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



#### **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 do this for you.*

### *Declaring a Method*

Line 5

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

is a comment indicating the purpose of lines 6–8 of the program. Line 6

```
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 in line 6; otherwise, the program will not execute.

Methods perform tasks and can return information when they complete their tasks. We’ll explain the purpose of keyword `static` in Section 3.2.5. 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 programs. The `String[] args` in parentheses is a required part of `main`’s declaration—we discuss this in Chapter 7.

The left brace at the end of line 6 begins the body of the method declaration. A corresponding right brace ends it (line 8). Line 7 is indented between the braces.



#### **Good Programming Practice 2.6**

*Indent the entire body of each method declaration one “level” between the braces that define the method’s body. This emphasizes the method’s structure and makes it easier to read.*

### Performing Output with `System.out.println`

Line 7

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

instructs the computer to perform an action—namely, to display the characters 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—later we'll show how to conveniently deal with long strings.

The `System.out` object—which is predefined for you—is known as the **standard output object**. It allows a program to display information in the command window from which the program executes. In Microsoft Windows, the command window is the **Command Prompt**. In UNIX/Linux/macOS, the command window is called a **terminal** 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 7 is the method's **argument**. 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 7, including `System.out.println`, the argument "Welcome to Java Programming!" in the parentheses and the semicolon (;), is called a **statement**. A method typically contains statements that perform its task. Most statements end with a semicolon.

### 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 8

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

indicates the closing brace of method `main`, and line 9

```
} // 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.

#### 2.2.1 Compiling the Application

We're now ready to compile and execute the program. We assume you're using the Java Development Kit's command-line tools, not an IDE. 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/macOS.

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 (or folders). On Windows, for example,

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

changes to the `fig02_01` directory. On UNIX/Linux/macOS, the command

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

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

```
javac Welcome1.java
```

If the program does not contain compilation errors, this command creates the file called `Welcome1.class` (known as `Welcome1`'s class file) 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

*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.*

When learning how to program, sometimes it's helpful to “break” a working program to get familiar with the compiler's error messages. These messages do not always state the exact problem in the code. When you encounter an error, it will give you an idea of what caused it. Try removing a semicolon or brace from the program of Fig. 2.1, then recompiling 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.*

Each compilation-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.

## 2.2.2 Executing the Application

Now that you've compiled the program, type the following command and press *Enter*:

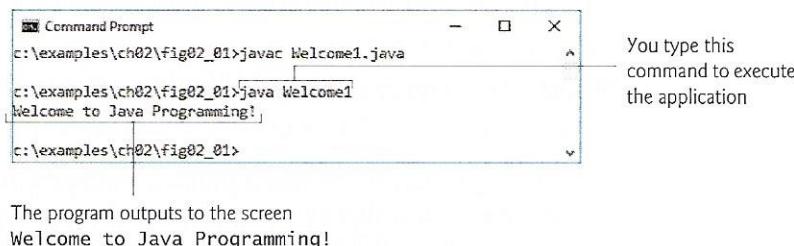
```
java Welcome1
```

to launch the JVM and load the `Welcome1.class` file. The command *omits* the `.class` filename extension; otherwise, the JVM will *not* execute the program. The JVM calls `Welcome1`'s `main` method. Next, line 7 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 7–8) 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 7–8.]

```

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

```

Welcome to Java Programming!

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

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 7–8 in 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 display-

ing its argument, `print` does *not* position the output cursor at the beginning of the next line—the next character the program displays will appear *immediately after* the last character that `print` displays. So, line 8 positions the first character in its argument (the letter “J”) immediately after the last character that line 7 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** (`\n`), 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 white space 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     // main method begins execution of Java application
6     public static void main(String[] args) {
7         System.out.println("Welcome\nto\nJava\nProgramming!");
8     } // end method main
9 } // end class Welcome3
```

```
Welcome
to
Java
Programming!
```

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

Line 7

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

displays four lines of text in the command window. Normally, the characters in a string are displayed *exactly* as they appear in the double quotes. However, 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**—\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 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/se8/html/jls-3.html#jls-3.10.6>

Escape sequence	Description
\n	Newline. Position the screen cursor at the beginning of the <i>next</i> 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 <i>current</i> line—do <i>not</i> advance to the next line. Any characters output after the carriage return <i>overwrite</i> 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

Method `System.out.printf` (*f* means “formatted”) displays *formatted* data. Figure 2.6 uses this to output on two lines the strings “Welcome to” and “Java Programming!”.

```

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

```

```
Welcome to
Java Programming!
```

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

Line 7

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

calls 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. Calling a method is also referred to as *invoking* a method.



### Good Programming Practice 2.7

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

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 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 per our convention 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     // main method begins execution of Java application
7     public static void main(String[] args) {
8         // create a Scanner to obtain input from the command window
9         Scanner input = new Scanner(System.in);
10
11        System.out.print("Enter first integer: "); // prompt
12        int number1 = input.nextInt(); // read first number from user
13
14        System.out.print("Enter second integer: "); // prompt
15        int number2 = input.nextInt(); // read second number from user
16
17        int sum = number1 + number2; // add numbers, then store total in sum
18
19        System.out.printf("Sum is %d%n", sum); // display sum
20    } // end method main
21 } // end class Addition

```

---

**Fig. 2.7** | Addition program that inputs two numbers then, displays their sum. (Part I of 2.)

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

**Fig. 2.7** | Addition program that inputs two numbers then, displays their sum. (Part 2 of 2.)

### 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 the program uses the predefined **Scanner** class (discussed shortly) from the package named **java.util**. The compiler then ensures that you use the class correctly.



#### Common Programming Error 2.5

*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 2.6

*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.*

### 2.5.2 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 variable’s *value* in memory. A variable 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 9 of **main**

```
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** (package **java.util**) 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 9 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 `ints`) that can be used in a program.



### 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 use the camel-case naming convention with a lowercase first letter—for example, `firstNumber`.*

## 2.5.3 Prompting the User for Input

Line 11

```
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`.



### Software Engineering Observation 2.1

*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.*

## 2.5.4 Declaring a Variable to Store an Integer and Obtaining an Integer from the Keyboard

The variable declaration statement in line 12

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

declares that variable `number1` holds data of type `int`—that is, *integer* values, which are whole numbers such as 72, -1127 and 0. The range of values for an `int` is -2,147,483,648 to +2,147,483,647. The `int` values you use in a program may not contain commas; however, for readability, you can place underscores in numbers. So 60\_000\_000 represents the `int` value 60,000,000.

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`).

The = in line 12 indicates that int variable number1 should be initialized in its declaration with the result of input.nextInt(). This uses the 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 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*.

### 2.5.5 Obtaining a Second Integer

Line 14

```
System.out.print("Enter second integer: "); // prompt  
prompts the user to enter the second integer. Line 15  
int number2 = input.nextInt(); // read second number from user
```

declares the int variable number2 and initializes it with a second integer read from the user at the keyboard.

### 2.5.6 Using Variables in a Calculation

Line 17

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

declares the int variable sum and initializes it with the result 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**, because it has *two operands*—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*. The value of the expression number1 + number2 is the *sum* of the numbers. Similarly, the value of the expression input.nextInt() (lines 12 and 15) is the integer typed by the user.



#### Good Programming Practice 2.10

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

### 2.5.7 Displaying the Calculation Result

After the calculation has been performed, line 19

```
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 also can be performed *inside* `printf` statements. We could have combined the statements at lines 17 and 19 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.

### 2.5.8 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/8/docs/api/index.html>

You can download it from the Additional Resources section at

<http://www.oracle.com/technetwork/java/javase/downloads>

Appendix F shows how to use this documentation.

### 2.5.9 Declaring and Initializing Variables in Separate Statements

Each variable must have a value *before* you can use the variable in a calculation (or other expression). The variable declaration statement in line 12 both declared `number1` *and* initialized it with a value entered by the user.

Sometimes you declare a variable in one statement, then initialize in another. For example, line 12 could have been written in two statements as

```
int number1; // declare the int variable number1
number1 = input.nextInt(); // assign the user's input to number1
```

The first statement declares `number1`, but does *not* initialize it. The second statement uses the assignment operator, `=`, to *assign* (that is, give) `number1` the value entered by the user. You can read this statement as “`number1` gets the value of `input.nextInt()`.” Everything to the *right* of the assignment operator, `=`, is always evaluated *before* the assignment is performed.

## 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 12) executes:

```
int 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*.

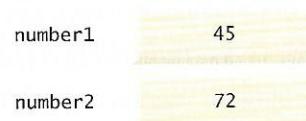


**Fig. 2.8** | Memory location showing the name and value of variable number1.

When the statement (line 15)

```
int 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.

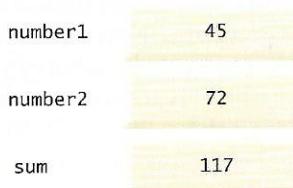


**Fig. 2.9** | Memory locations after storing values for number1 and number2.

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 17)

```
int 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	+	$f + 7$	<code>f + 7</code>
Subtraction	-	$p - c$	<code>p - c</code>
Multiplication	*	$bm$	<code>b * m</code>
Division	/	$x / y$ or $\frac{x}{y}$ or $x \div y$	<code>x / y</code>
Remainder	%	$r \bmod s$	<code>r % s</code>

**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 computers. 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 arithmetic operators 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*.<sup>1</sup> When we say that operators are applied from left to right, we're referring to their associativity. Some 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	Evaluated first. If there are several operators of this type, they're evaluated from <i>left to right</i> .
/	Division	
%	Remainder	
+	Addition	Evaluated next. If there are several operators of this type, they're evaluated from <i>left to right</i> .
-	Subtraction	
=	Assignment	Evaluated last.

**Fig. 2.12** | Precedence of arithmetic operators.

### Sample Algebraic and Java Expressions

Let's consider several sample expressions. Each example shows an algebraic expression and its Java equivalent. The following is an example of an average of five terms:

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

$$\text{Java: } m = (a + b + c + d + e) / 5;$$

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 to the different expression

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

Here's an example of the equation of a straight line:

$$\text{Algebra: } y = mx + b$$

$$\text{Java: } y = m * x + b;$$

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:

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

$$\text{Java: } z = p * r \% q + w / x - y;$$

6      1      2      4      3      5

- 
- We use simple examples to explain the *order of evaluation*. Subtle order-of-evaluation issues occur in the more complex expressions. For more information, see Chapter 15 of *The Java™ Language Specification* (<https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html>).

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.

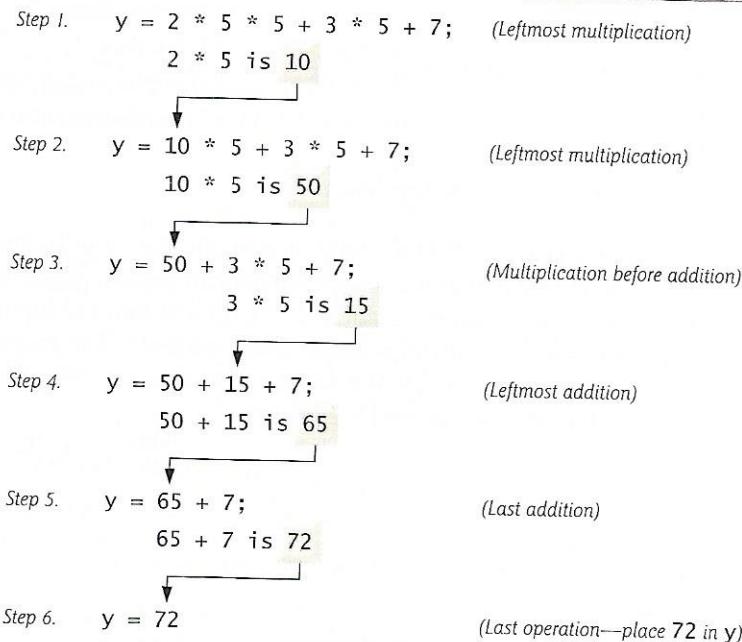
### Evaluation of a Second-Degree Polynomial

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 5.4 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.



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

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

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

## 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 an *if* statement's condition is *true*, its body executes. If the condition is *false*, its body does not execute.

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
<i>Equality operators</i>			
=	<code>==</code>	<code>x == y</code>	<code>x</code> is equal to <code>y</code>
≠	<code>!=</code>	<code>x != y</code>	<code>x</code> is not equal to <code>y</code>
<i>Relational operators</i>			
>	<code>&gt;</code>	<code>x &gt; y</code>	<code>x</code> is greater than <code>y</code>
<	<code>&lt;</code>	<code>x &lt; y</code>	<code>x</code> is less than <code>y</code>
≥	<code>&gt;=</code>	<code>x &gt;= y</code>	<code>x</code> is greater than or equal to <code>y</code>
≤	<code>&lt;=</code>	<code>x &lt;= y</code>	<code>x</code> is less than or equal to <code>y</code>

**Fig. 2.14** | Equality and relational operators.

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. We show three sample outputs for different values entered by the user.

```

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     // main method begins execution of Java application
8     public static void main(String[] args) {
9         // create Scanner to obtain input from command line
10        Scanner input = new Scanner(System.in);

```

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

```
11
12     System.out.print("Enter first integer: "); // prompt
13     int number1 = input.nextInt(); // read first number from user
14
15     System.out.print("Enter second integer: "); // prompt
16     int number2 = input.nextInt(); // read second number from user
17
18     if (number1 == number2)
19         System.out.printf("%d == %d%n", number1, number2);
20     }
21
22     if (number1 != number2) {
23         System.out.printf("%d != %d%n", number1, number2);
24     }
25
26     if (number1 < number2) {
27         System.out.printf("%d < %d%n", number1, number2);
28     }
29
30     if (number1 > number2) {
31         System.out.printf("%d > %d%n", number1, number2);
32     }
33
34     if (number1 <= number2) {
35         System.out.printf("%d <= %d%n", number1, number2);
36     }
37
38     if (number1 >= number2) {
39         System.out.printf("%d >= %d%n", number1, number2);
40     }
41 } // end method main
42 } // 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.)

Class Comparison's main method (lines 8–41) begins the execution of the program.  
Line 10

```
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 12–13

```
System.out.print("Enter first integer: "); // prompt
int 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 the int variable number1.

Lines 15–16

```
System.out.print("Enter second integer: "); // prompt
int 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 the int variable number2.

Lines 18–20

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

compare the values of variables number1 and number2 to test for equality. If the values are equal, the statement in line 19 displays a line of text indicating that the numbers are equal. The if statements starting in lines 22, 26, 30, 34 and 38 compare number1 and number2 using the operators !=, <, >, <= and >=, respectively. If the conditions are true in one or more of those if statements, the corresponding body statement displays an appropriate line of text.

Each if statement in Fig. 2.15 contains a single body statement that's indented. Also notice that we've enclosed each body statement in a pair of braces, { }, creating what's called a compound statement or a block.



### Good Programming Practice 2.11

*Indent the statement(s) in the body of an if statement to enhance readability. IDEs typically do this for you, allowing you to specify the indent size.*



### Error-Prevention Tip 2.4

*You don't need to use braces, { }, around single-statement bodies, but you must include the braces around multiple-statement bodies. You'll see later that forgetting to enclose multiple-statement bodies in braces leads to errors. To avoid errors, as a rule, always enclose an if statement's body statement(s) in braces.*



### Common Programming Error 2.7

*Placing a semicolon immediately after the right parenthesis after the condition in an if statement is often a logic error (although not a syntax error). The semicolon causes the body of the if statement to be empty, so the if statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the if statement always executes, often causing the program to produce incorrect results.*

### 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 2.5

*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`.

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

**Fig. 2.16** | Precedence and associativity of operators discussed.



### 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.*

## 2.9 Wrap-Up

In this chapter, you learned many important features of Java, including displaying data on the screen in a command window, inputting data from the keyboard, performing calculations and making decisions. The applications presented here introduced you to many basic programming concepts. As you'll see in Chapter 3, Java applications typically contain just a few lines of code in method `main`—these statements normally create the objects that perform the work of the application. In Chapter 3, you'll learn how to implement your own classes and use objects of those classes in applications.

## Summary

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

- A Java application (p. 36) executes when you use the `java` command to launch the JVM.
- Comments (p. 37) document programs and improve their readability. The compiler ignores them.
- An end-of-line comment begins with `//` and terminates at the end of the line on which it appears.
- Traditional comments (p. 37) can be spread over several lines and are delimited by `/*` and `*/`.
- Javadoc comments (p. 37), delimited by `/**` and `*/`, enable you to embed program documentation in your code. The `javadoc` program generates web pages based on these comments.
- A syntax error (p. 37) 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. 38). White space makes programs easier to read and is normally ignored by the compiler.
- Keywords (p. 38) are reserved for use by Java and are always spelled with all lowercase letters.
- Keyword `class` (p. 38) 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.
- A `public` (p. 38) class declaration must be saved in a file with the same name as the class followed by the “`.java`” filename extension.
- Java is case sensitive (p. 38)—that is, uppercase and lowercase letters are distinct.
- The body of every class declaration (p. 39) is delimited by braces, `{` and `}`.
- Method `main` (p. 39) 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. 39) indicates that a method will perform a task but return no information.
- Statements instruct the computer to perform actions.
- A string (p. 40) in double quotes is sometimes called a character string or a string literal.
- The standard output object (`System.out`; p. 40) displays characters in the command window.
- Method `System.out.println` (p. 40) displays its argument (p. 40) in the command window followed by a newline character to position the output cursor to the beginning of the next line.

### *Section 2.2.1 Compiling the Application*

- You compile a program with the command `javac`. If the program contains no syntax errors, a class file (p. 41) containing the Java bytecodes that represent the application is created. These bytecodes are interpreted by the JVM when you execute the program.

### *Section 2.2.2 Executing the Application*

- To run an application, type `java` followed by the name of the class that contains method `main`.

### *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)—\n (p. 43) represents the newline character.

### ***Section 2.4 Displaying Text with printf***

- `System.out.printf` method (p. 44; 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. 45) is a placeholder for a string.
- The %n format specifier (p. 45) 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 its argument is portable across operating systems.

### ***Section 2.5.1 import Declarations***

- An `import` declaration (p. 46) helps the compiler locate a class that’s used in a program.
- Java’s rich set of predefined classes are grouped into packages (p. 46)—named groups of classes. These are referred to as the Java class library, or the Java Application Programming Interface (Java API; p. 46).

### ***Section 2.5.2 Declaring and Creating a Scanner to Obtain User Input from the Keyboard***

- 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. 46) 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. 46) 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.

### ***Section 2.5.3 Prompting the User for Input***

- A prompt (p. 47) directs the user to take a specific action.

### ***Section 2.5.4 Declaring a Variable to Store an Integer and Obtaining an Integer from 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.
- The `int` values you use in a program may not contain commas; however, for readability, you can place underscores in numbers (e.g., 60\_000\_000).
- Types `float` and `double` (p. 47) specify real numbers with decimal points, such as -11.19 and 3.4.
- 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.
- `Scanner` method `nextInt` obtains an integer for use in a program.

**Section 2.5.6 Using Variables in a Calculation**

- Portions of statements that have values are called expressions (p. 48).

**Section 2.5.7 Displaying the Calculation Result**

- The format specifier %d (p. 48) is a placeholder for an int value.

**Section 2.5.9 Declaring and Initializing Variables in Separate Statements**

- A variable must be assigned a value before it's used in a program.
- The assignment operator, = (p. 49), enables the program to give a value to a variable.

**Section 2.6 Memory Concepts**

- Variable names (p. 49) 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. 50) 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. 51), 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. 51).
- 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. You must include braces around multiple-statement bodies.

**Self-Review Exercises**

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

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

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

- Comments cause the computer to display 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.
  - f) The identifier `_` (underscore) is valid in Java 9.
- 2.3** Write statements to accomplish each of the following tasks:
- a) Declare variables `c`, `thisIsAVariable`, `q76354` and `number` to be of type `int` and initialize each to 0.
  - 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) Prompt the user to enter the first integer.
  - d) Read the first integer from the user and store it in the `int` variable `x`.
  - e) Prompt the user to enter the second integer.
  - f) Read the second integer from the user and store it in the `int` variable `y`.
  - g) Prompt the user to enter the third integer.
  - h) Read the third integer from the user and store it in the `int` variable `z`.
  - i) Compute the product of the three integers contained in variables `x`, `y` and `z`, and store the result in the `int` variable `result`.
  - j) 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** The answers to Self-Review Exercise 2.2 are:
- 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 % have higher precedence than operators + and -.
- f) False. As of Java 9, \_ (underscore) by itself is no longer a valid identifier.

**2.3** The answers to Self-Review Exercise 2.3 are:

- a) 

```
int c = 0;
    int thisIsAVariable = 0;
    int q76354 = 0;
    int number = 0;
```
- b) 

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

```
int 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** The answers to Self-Review Exercise 2.4 are:

- a) Error: Semicolon after the right parenthesis of the condition ( $c < 7$ ) in the if. As a result, the output statement executes regardless of whether the condition in the if is true.  
Correction: Remove the semicolon after the right parenthesis.
- b) Error: The relational operator => is incorrect.  
Correction: Change => to >=.

**2.5** The answers to Self-Review Exercise 2.5 are:

- a) *// Calculate the product of three integers*
- b) 

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

```
System.out.print("Enter first integer: ");
```
- d) 

```
int x = input.nextInt();
```
- e) 

```
System.out.print("Enter second integer: ");
```
- f) 

```
int y = input.nextInt();
```
- g) 

```
System.out.print("Enter third integer: ");
```
- h) 

```
int z = input.nextInt();
```
- i) 

```
int result = x * y * z;
```
- j) 

```
System.out.printf("Product is %d%n", result);
```

**2.6** The answer to Self-Review Exercise 2.6 is:

---

```

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     public static void main(String[] args) {
7         // create Scanner to obtain input from command window
8         Scanner input = new Scanner(System.in);
9
10        System.out.print("Enter first integer: "); // prompt for input
11        int x = input.nextInt(); // read first integer
12
13        System.out.print("Enter second integer: "); // prompt for input
14        int y = input.nextInt(); // read second integer
15
16        System.out.print("Enter third integer: "); // prompt for input
17        int z = input.nextInt(); // read third integer
18

```

---

```

19     int result = x * y * z; // calculate product of numbers
20
21     System.out.printf("Product is %d%n", result);
22 } // end method main
23 } // 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:
- \_\_\_\_\_ are used to document a program and improve its readability.
  - A decision can be made in a Java program with a(n) \_\_\_\_\_.
  - The arithmetic operators with the same precedence as multiplication are \_\_\_\_\_ and \_\_\_\_\_.
  - When parentheses in an arithmetic expression are nested, the \_\_\_\_\_ set of parentheses is evaluated first.
  - 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:
- Display the message "Enter an integer: ", leaving the cursor on the same line.
  - Assign the product of variables b and c to the int variable a.
  - 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.
- Java operators are evaluated from left to right.
  - The following are all valid variable names: \_under\_bar\_, m928134, t5, j7, her\_sales\$, his\_\$account\_total, a, b\$, c, z and z2.
  - A valid Java arithmetic expression with no parentheses is evaluated from left to right.
  - The following are all invalid variable names: 3g, 87, 67h2, h22 and 2h.
- 2.10** Assuming that  $x = 2$  and  $y = 3$ , what does each of the following statements display?
- `System.out.printf("x = %d%n", x);`
  - `System.out.printf("Value of %d + %d is %d%n", x, x, (x + x));`
  - `System.out.printf("x =");`
  - `System.out.printf("%d = %d%n", (x + y), (y + x));`
- 2.11** Which of the following Java statements contain variables whose values are modified?
- `int p = i + j + k + 7;`
  - `System.out.println("variables whose values are modified");`
  - `System.out.println("a = 5");`
  - `int value = input.nextInt();`
- 2.12** Given that  $y = ax^3 + 7$ , which of the following are correct Java statements for this equation?
- `int y = a * x * x * x + 7;`
  - `int y = a * x * x * x * (x + 7);`
  - `int y = (a * x) * x * (x + 7);`
  - `int y = (a * x) * x * x + 7;`
  - `int y = a * (x * x * x) + 7;`
  - `int 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:

- int x = 7 + 3 \* 6 / 2 - 1;
- int x = 2 % 2 + 2 \* 2 - 2 / 2;
- int 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:

- Use one `System.out.println` statement.
- Use four `System.out.print` statements.
- 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 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  $\pi$ . Use the techniques shown in Fig. 2.7. [Note: You may also use the predefined constant `Math.PI` for the value of  $\pi$ . 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):

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

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 4.) 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 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 BMI categories and their values from the National Heart Lung and Blood Institute

[http://www.nhlbi.nih.gov/health/educational/lose\\_wt/BMI/bmicalc.htm](http://www.nhlbi.nih.gov/health/educational/lose_wt/BMI/bmicalc.htm)

so the user can evaluate his/her BMI.

[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 `doubles`, they'll both produce numbers with decimal points—these are called “floating-point” numbers.]

**2.34 (World Population Growth Calculator)** Search the Internet 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.