**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.

**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.

<code>

public class Welcome1

{

// main method begins execution of Java application

public static void main(String[] args)

{

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

} // end method main

} // end class Welcome1

</code>

**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.

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

<code> // Fig. 2.1: Welcome1.java </code>

**Performing Output with System.out.println**

<code>System.out.println("Welcome to Java Programming!"); </code>

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, it will give you

an idea of what caused it. [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.]

**Scanner Object for Receiving Input from the User**

You can Create a Scanner object named input for inputting the name from the user.then prompt the user to enter a name. and uses the Scanner object’s nextLine method to

read the name from the user and assign it to the local variable theName. You type the name

and press Enter to submit it to the program. Pressing Enter inserts a newline character after

the characters you typed. Method nextLine reads characters (including white-space characters,

such as the blank in "Jane Green") until it encounters the newline, then returns a

String containing the characters up to, but not including, the newline, which is discarded.

Class Scanner provides various other input methods, as you’ll see throughout the

book. A method similar to nextLine—named next—reads the next word. When you press

Enter after typing some text, method next reads characters until it encounters a white-space

character (such as a space, tab or newline), then returns a String containing the characters

up to, but not including, the white-space character, which is discarded. All information

after the first white-space character is not lost—it can be read by subsequent statements that

call the Scanner’s methods later in the program.

**Instantiating an Object—Keyword new and Constructors**

create an Account object and assigns it to variable myAccount of type Account.

Variable myAccount is initialized with the result of the class instance creation expression

new Account(). Keyword new creates a new object of the specified class—in this case, Account.

The parentheses to the right of Account are required. As you’ll learn in Section 3.4,

those parentheses in combination with a class name represent a call to a constructor, which

is similar to a method but is called implicitly by the new operator to initialize an object’s

instance variables when the object is created. In Section 3.4, you’ll see how to place an argument

in the parentheses to specify an initial value for an Account object’s name instance

variable—you’ll enhance class Account to enable this. For now, we simply leave the parentheses

empty. Line 10 contains a class instance creation expression for a Scanner object the expression initializes the Scanner with System.in, which tells the Scanner where to

read the input from (i.e., the keyboard).

**Software Engineering with private Instance Variables and**

**public set and get Methods**

As you’ll see, through the use of *set* and *get* methods, you can *validate* attempted modifications

to private data and control how that data is presented to the caller—these are

compelling software engineering benefits. We’ll discuss this in more detail in Section 3.5.

If the instance variable were public, any **client** of the class—that is, any other class

that calls the class’s methods—could see the data and do whatever it wanted with it,

including setting it to an *invalid* value.

You might think that even though a client of the class cannot directly access a private

instance variable, the client can do whatever it wants with the variable through public *set*

and *get* methods. You would think that you could peek at the private data any time with

the public *get* method and that you could modify the private data at will through the

public *set* method. But *set* methods can be programmed to *validate* their arguments and

reject any attempts to *set* the data to bad values, such as a negative body temperature, a day

in March out of the range 1 through 31, a product code not in the company’s product

catalog, etc. And a *get* method can present the data in a different form. For example, a

Grade class might store a grade as an int between 0 and 100, but a getGrade method

might return a letter grade as a String, such as "A" for grades between 90 and 100, "B"

for grades between 80 and 89, etc. Tightly controlling the access to and presentation ofprivate data can greatly reduce errors, while increasing the robustness and security of

your programs.

Declaring instance variables with access modifier private is known as *data hiding* or

*information hiding*. When a program creates (instantiates) an object of class Account, variable

name is *encapsulated* (hidden) in the object and can be accessed only by methods of

the object’s class.

**Primitive Types vs. Reference Types**

Java’s types are divided into primitive types and **reference types**. In Chapter 2, you worked

with variables of type int—one of the primitive types. The other primitive types are

boolean, byte, char, short, long, float and double, each of which we discuss in this

book—these are summarized in Appendix D. All nonprimitive types are *reference types*, so

classes, which specify the types of objects, are reference types.

A primitive-type variable can hold exactly *one* value of its declared type at a time. For

example, an int variable can store one integer at a time. When another value is assigned

to that variable, the new value replaces the previous one—which is *lost*.

Recall that local variables are *not* initialized by default. Primitive-type instance variables

*are* initialized by default—instance variables of types byte, char, short, int, long,

float and double are initialized to 0, and variables of type boolean are initialized to

false. You can specify your own initial value for a primitive-type variable by assigning the

variable a value in its declaration, as in

Programs use variables of reference types (normally called **references**) to store the

*addresses* of objects in the computer’s memory. Such a variable is said to **refer to an object**

in the program. *Objects* that are referenced may each contain *many* instance variables.

creates an object of class Scanner, then assigns to the variable input a *reference* to that

Scanner object. creates an object of class Account, then assigns to the variable myAccount a *reference* to that

Account object. *Reference-type instance variables, if not explicitly initialized, are initialized*

*by default to the value null*—which represents a “reference to nothing.” That’s why the

first call to getName in line 16 of Fig. 3.2 returns null—the value of name has *not* yet been

set, so the *default initial value* null is returned.

To call methods on an object, you need a reference to the object. In Fig. 3.2, the statements

in method main use the variable myAccount to call methods getName (lines 16 and

26) and setName (line 21) to interact with the Account object. Primitive-type variables do

*not* refer to objects, so such variables *cannot* be used to call methods.