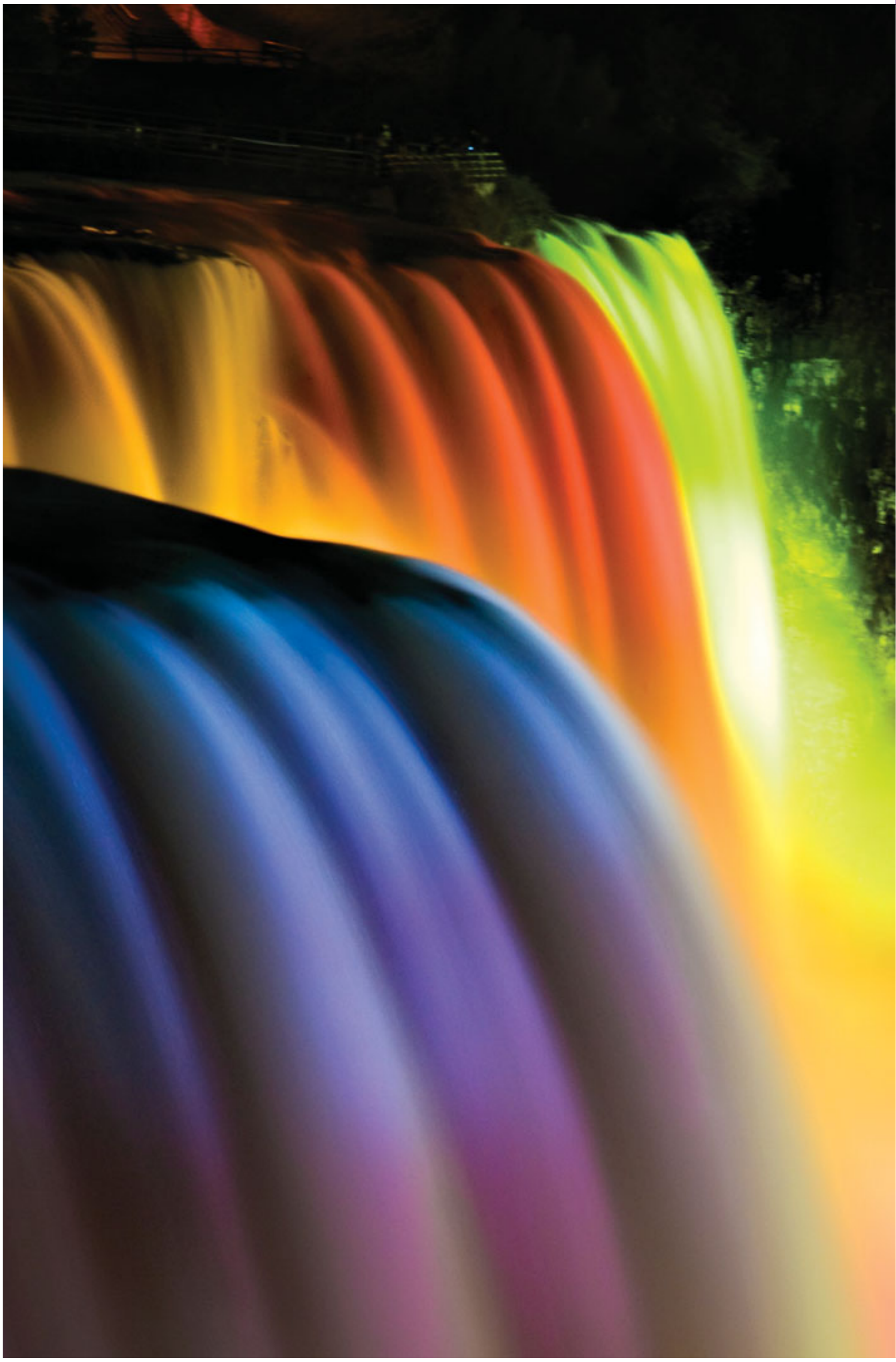
Introduction to Classes, Objects, Methods and Strings

3



Nothing can have value without being an object of utility.
—Karl Marx

Your public servants serve you right.
—Adlai E. Stevenson

You'll see something new. Two things. And I call them Thing One and Thing Two.
—Dr. Theodor Seuss Geisel

Objectives

In this chapter you'll learn:

- How to declare a class and use it to create an object.
- How to implement a class's behaviors as methods.
- How to implement a class's attributes as instance variables and properties.
- How to call an object's methods to make them perform their tasks.
- What instance variables of a class and local variables of a method are.
- How to use a constructor to initialize an object's data.
- The differences between primitive and reference types.

Outline

3.1 Introduction

We introduced the basic terminology and concepts of object-oriented programming in Section 1.6. In this chapter, we present a simple framework for organizing object-oriented applications in Java. Typically, the applications you develop in this book will consist of two or more classes. If you become part of a development team in industry, you might work on applications that contain hundreds, or even thousands, of classes.

First, we motivate the notion of classes with a real-world example. Then we present five applications to demonstrate creating and using your own classes. The first four of these begin our case study on developing a grade book class that instructors can use to maintain student test scores. This case study is enhanced in Chapters 4, 5 and 7. The last example introduces floating-point numbers—that is, numbers containing decimal points—in a bank account class that maintains a customer's balance.

3.2 Declaring a Class with a Method and Instantiating an Object of a Class

In Sections 2.5 and 2.8, you created an object of the *existing* class `Scanner`, then used that object to read data from the keyboard. In this section, you'll create a *new* class, then use it to create an object. We begin by delcaring classes `GradeBook` (Fig. 3.1) and `GradeBookTest` (Fig. 3.2). Class `GradeBook` (declared in the file `GradeBook.java`) will be used to display a message on the screen (Fig. 3.2) welcoming the instructor to the grade book application. Class `GradeBookTest` (declared in the file `GradeBookTest.java`) is an application class in which the `main` method will create and use an object of class `GradeBook`. *Each class declaration that begins with keyword* `public` *must be stored in a file having the same name as the class and ending with the* `.java` *file-name extension.* Thus, classes `GradeBook` and `GradeBookTest` must be declared in *separate* files, because each class is declared `public`.

Class GradeBook

The `GradeBook` class declaration (Fig. 3.1) contains a `displayMessage` method (lines 7–10) that displays a message on the screen. We'll need to make an object of this class and call its method to execute line 9 and display the message.

The *class declaration* begins in line 4. The keyword `public` is an **access modifier**. For now, we'll simply declare every class `public`. Every class declaration contains keyword

```
1  // Fig. 3.1: GradeBook.java
2  // Class declaration with one method.
3
4  public class GradeBook
5  {
6     // display a welcome message to the GradeBook user
7     public void displayMessage()
8     {
9        System.out.println( "Welcome to the Grade Book!" );
10    } // end method displayMessage
11 } // end class GradeBook
```

Fig. 3.1 | Class declaration with one method.

`class` followed immediately by the class's name. Every class's body is enclosed in a pair of left and right braces, as in lines 5 and 11 of class `GradeBook`.

In Chapter 2, each class we declared had one method named `main`. Class `GradeBook` also has one method—`displayMessage` (lines 7–10). Recall that `main` is a special method that's *always* called automatically by the Java Virtual Machine (JVM) when you execute an application. Most methods do not get called automatically. As you'll soon see, you must call method `displayMessage` explicitly to tell it to perform its task.

The method declaration begins with keyword `public` to indicate that the method is "available to the public"—it can be called from methods of other classes. Next is the method's **return type**, which specifies the type of data the method returns to its caller after performing its task. The return type `void` indicates that this method will perform a task but will *not* return (i.e., give back) any information to its **calling method**. You've used methods that return information—for example, in Chapter 2 you used `Scanner` method `nextInt` to input an integer typed by the user at the keyboard. When `nextInt` reads a value from the user, it returns that value for use in the program.

The name of the method, `displayMessage`, follows the return type. By convention, method names begin with a lowercase first letter and subsequent words in the name begin with a capital letter. The parentheses after the method name indicate that this is a method. Empty parentheses, as in line 7, indicate that this method does not require additional information to perform its task. Line 7 is commonly referred to as the **method header**. Every method's body is delimited by left and right braces, as in lines 8 and 10.

The body of a method contains one or more statements that perform the method's task. In this case, the method contains one statement (line 9) that displays the message `"Welcome to the Grade Book!"` followed by a newline (because of `println`) in the command window. After this statement executes, the method has completed its task.

Class `GradeBookTest`

Next, we'd like to use class `GradeBook` in an application. As you learned in Chapter 2, method `main` begins the execution of *every* application. A class that contains method `main` begins the execution of a Java application. Class `GradeBook` is *not* an application because it does *not* contain `main`. Therefore, if you try to execute `GradeBook` by typing `java GradeBook` in the command window, an error will occur. This was not a problem in Chapter 2, because every class you declared had a `main` method. To fix this problem, we must either declare a separate class that contains a `main` method or place a `main` method in class `Grade-`

Book. To help you prepare for the larger programs you'll encounter later in this book and in industry, we use a separate class (GradeBookTest in this example) containing method main to test each new class we create in this chapter. Some programmers refer to such a class as a *driver class*.

The GradeBookTest class declaration (Fig. 3.2) contains the main method that will control our application's execution. The GradeBookTest class declaration begins in line 4 and ends in line 15. The class, like many that begin an application's execution, contains *only* a main method.

```
1  // Fig. 3.2: GradeBookTest.java
2  // Creating a GradeBook object and calling its displayMessage method.
3
4  public class GradeBookTest
5  {
6     // main method begins program execution
7     public static void main( String[] args )
8     {
9        // create a GradeBook object and assign it to myGradeBook
10       GradeBook myGradeBook = new GradeBook();
11
12       // call myGradeBook's displayMessage method
13       myGradeBook.displayMessage();
14    } // end main
15 } // end class GradeBookTest
```

```
Welcome to the Grade Book!
```

Fig. 3.2 | Creating a GradeBook object and calling its displayMessage method.

Lines 7–14 declare method main. A key part of enabling the JVM to locate and call method main to begin the application's execution is the static keyword (line 7), which indicates that main is a static method. *A static method is special, because you can call it without first creating an object of the class in which the method is declared.* We discuss static methods in Chapter 6, Methods: A Deeper Look.

In this application, we'd like to call class GradeBook's displayMessage method to display the welcome message in the command window. Typically, you cannot call a method that belongs to another class until you create an object of that class, as shown in line 10. We begin by declaring variable myGradeBook. The variable's type is GradeBook—the class we declared in Fig. 3.1. Each new *class* you create becomes a new *type* that can be used to declare variables and create objects. You can declare new class types as needed; this is one reason why Java is known as an **extensible language**.

Variable myGradeBook is initialized (line 10) with the result of the **class instance creation expression** new GradeBook(). Keyword **new** creates a new object of the class specified to the right of the keyword (i.e., GradeBook). The parentheses to the right of GradeBook are required. As you'll learn in Section 3.6, those parentheses in combination with a class name represent a call to a **constructor**, which is similar to a method but is used only at the time an object is *created* to *initialize* the object's data. You'll see that data can be placed in the parentheses to specify *initial values* for the object's data. For now, we simply leave the parentheses empty.

Just as we can use object `System.out` to call its methods `print`, `printf` and `println`, we can use object `myGradeBook` to call its method `displayMessage`. Line 13 calls the method `displayMessage` (lines 7–10 of Fig. 3.1) using `myGradeBook` followed by a **dot separator** (`.`), the method name `displayMessage` and an empty set of parentheses. This call causes the `displayMessage` method to perform its task. This method call differs from those in Chapter 2 that displayed information in a command window—each of those method calls provided arguments that specified the data to display. At the beginning of line 13, "`myGradeBook.`" indicates that `main` should use the `myGradeBook` object that was created in line 10. Line 7 of Fig. 3.1 indicates that method `displayMessage` has an *empty parameter list*—that is, `displayMessage` does *not* require additional information to perform its task. For this reason, the method call (line 13 of Fig. 3.2) specifies an empty set of parentheses after the method name to indicate that *no arguments* are being passed to method `displayMessage`. When method `displayMessage` completes its task, method `main` continues executing at line 14. This is the end of method `main`, so the program terminates.

Any class can contain a `main` method. The JVM invokes the `main` method *only* in the class used to execute the application. If an application has multiple classes that contain `main`, the one that's invoked is the one in the class named in the `java` command.

Compiling an Application with Multiple Classes

You must compile the classes in Fig. 3.1 and Fig. 3.2 before you can execute the application. First, change to the directory that contains the application's source-code files. Next, type the command

```
javac GradeBook.java GradeBookTest.java
```

to compile *both* classes at once. If the directory containing the application includes only this application's files, you can compile *all* the classes in the directory with the command

```
javac *.java
```

The asterisk (*) in `*.java` indicates that *all* files in the current directory that end with the file-name extension "`.java`" should be compiled.

UML Class Diagram for Class `GradeBook`

Figure 3.3 presents a **UML class diagram** for class `GradeBook` of Fig. 3.1. In the UML, each class is modeled in a class diagram as a rectangle with three compartments. The top compartment contains the name of the class centered horizontally in boldface type. The middle compartment contains the class's attributes, which correspond to instance variables (discussed in Section 3.4) in Java. In Fig. 3.3, the middle compartment is empty, because this `GradeBook` class does *not* have any attributes. The bottom compartment contains the

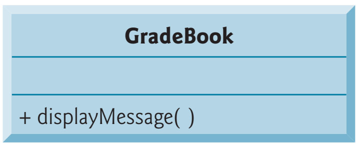


Fig. 3.3 | UML class diagram indicating that class `GradeBook` has a `public` `displayMessage` operation.

class's **operations**, which correspond to methods in Java. The UML models operations by listing the operation name preceded by an access modifier (in this case +) and followed by a set of parentheses. Class `GradeBook` has one method, `displayMessage`, so the bottom compartment of Fig. 3.3 lists one operation with this name. Method `displayMessage` does *not* require additional information to perform its tasks, so the parentheses following the method name in the class diagram are *empty*, just as they were in the method's declaration in line 7 of Fig. 3.1. The plus sign (+) in front of the operation name indicates that `displayMessage` is a public operation in the UML (i.e., a `public` method in Java). We'll often use UML class diagrams to summarize a class's attributes and operations.

3.3 Declaring a Method with a Parameter

In our car analogy from Section 1.6, we discussed the fact that pressing a car's gas pedal sends a *message* to the car to *perform a task*—to go faster. But *how fast* should the car accelerate? As you know, the farther down you press the pedal, the faster the car accelerates. So the message to the car actually includes the *task to perform* and *additional information* that helps the car perform the task. This additional information is known as a **parameter**—the value of the parameter helps the car determine how fast to accelerate. Similarly, a method can require one or more parameters that represent additional information it needs to perform its task. Parameters are defined in a comma-separated **parameter list**, which is located inside the parentheses that follow the method name. Each parameter must specify a *type* and a variable name. The parameter list may contain any number of parameters, including none at all. Empty parentheses following the method name (as in Fig. 3.1, line 7) indicate that a method does *not* require any parameters.

Arguments to a Method

A method call supplies values—called *arguments*—for each of the method's parameters. For example, the method `System.out.println` requires an argument that specifies the data to output in a command window. Similarly, to make a deposit into a bank account, a `deposit` method specifies a parameter that represents the deposit amount. When the `deposit` method is called, an argument value representing the deposit amount is assigned to the method's parameter. The method then makes a deposit of that amount.

Class Declaration with a Method That Has One Parameter

We now declare class `GradeBook` (Fig. 3.4) with a `displayMessage` method that displays the course name as part of the welcome message. (See the sample execution in Fig. 3.5.) The new method requires a parameter that represents the course name to output.

Before discussing the new features of class `GradeBook`, let's see how the new class is used from the `main` method of class `GradeBookTest` (Fig. 3.5). Line 12 creates a `Scanner` named `input` for reading the course name from the user. Line 15 creates the `GradeBook` object `myGradeBook`. Line 18 prompts the user to enter a course name. Line 19 reads the name from the user and assigns it to the `nameOfCourse` variable, using `Scanner` method **`nextLine`** to perform the input. The user types the course name and presses *Enter* to submit the course name to the program. Pressing *Enter* inserts a newline character at the end of the characters typed by the user. Method `nextLine` reads characters typed by the user until it encounters the newline character, then returns a `String` containing the characters up to, but *not* including, the newline. The newline character is *discarded*.

```java
// Fig. 3.4: GradeBook.java
// Class declaration with one method that has a parameter.

public class GradeBook
{
   // display a welcome message to the GradeBook user
   public void displayMessage( String courseName )
   {
      System.out.printf( "Welcome to the grade book for\n%s!\n",
         courseName );
   } // end method displayMessage
} // end class GradeBook
```

Fig. 3.4 | Class declaration with one method that has a parameter.

```java
// Fig. 3.5: GradeBookTest.java
// Create GradeBook object and pass a String to
// its displayMessage method.
import java.util.Scanner; // program uses Scanner

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String[] args )
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      // create a GradeBook object and assign it to myGradeBook
      GradeBook myGradeBook = new GradeBook();

      // prompt for and input course name
      System.out.println( "Please enter the course name:" );
      String nameOfCourse = input.nextLine(); // read a line of text
      System.out.println(); // outputs a blank line

      // call myGradeBook's displayMessage method
      // and pass nameOfCourse as an argument
      myGradeBook.displayMessage( nameOfCourse );
   } // end main
} // end class GradeBookTest
```

```
Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

Fig. 3.5 | Create a `GradeBook` object and pass a `String` to its `displayMessage` method.

Class `Scanner` also provides a similar method—**`next`**—that reads individual words. When the user presses *Enter* after typing input, 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 other statements that call the `Scanner`'s methods later in the program. Line 20 outputs a blank line.

Line 24 calls `myGradeBooks`'s `displayMessage` method. The variable `nameOfCourse` in parentheses is the *argument* that's passed to method `displayMessage` so that the method can perform its task. The value of variable `nameOfCourse` in `main` becomes the value of method `displayMessage`'s *parameter* `courseName` in line 7 of Fig. 3.4. When you execute this application, notice that method `displayMessage` outputs the name you type as part of the welcome message (Fig. 3.5).

More on Arguments and Parameters

In Fig. 3.4, `displayMessage`'s parameter list (line 7) declares one parameter indicating that the method requires a `String` to perform its task. When the method is called, the argument value in the call is assigned to the corresponding parameter (`courseName`) in the method header. Then, the method body uses the value of the `courseName` parameter. Lines 9–10 of Fig. 3.4 display parameter `courseName`'s value, using the `%s` format specifier in `printf`'s format string. The parameter variable's name (`courseName` in Fig. 3.4, line 7) can be the *same or different* from the argument variable's name (`nameOfCourse` in Fig. 3.5, line 24).

The number of arguments in a method call *must* match the number of parameters in the parameter list of the method's declaration. Also, the argument types in the method call must be "consistent with" the types of the corresponding parameters in the method's declaration. (As you'll learn in Chapter 6, an argument's type and its corresponding parameter's type are not always required to be *identical*.) In our example, the method call passes one argument of type `String` (`nameOfCourse` is declared as a `String` in line 19 of Fig. 3.5) and the method declaration specifies one parameter of type `String` (`courseName` is declared as a `String` in line 7 of Fig. 3.4). So in this example the type of the argument in the method call exactly matches the type of the parameter in the method header.

*Updated UML Class Diagram for Class **GradeBook***

The UML class diagram of Fig. 3.6 models class `GradeBook` of Fig. 3.4. Like Fig. 3.1, this `GradeBook` class contains `public` operation `displayMessage`. However, this version of `displayMessage` has a parameter. The UML models a parameter a bit differently from Java by listing the parameter name, followed by a colon and the parameter type in the parentheses following the operation name. The UML has its own data types similar to those of Java (but, as you'll see, not all the UML data types have the same names as the corresponding Java types). The UML type `String` does correspond to the Java type `String`. `GradeBook` method `displayMessage` (Fig. 3.4) has a `String` parameter named `courseName`, so Fig. 3.6 lists `courseName : String` between the parentheses following `displayMessage`.

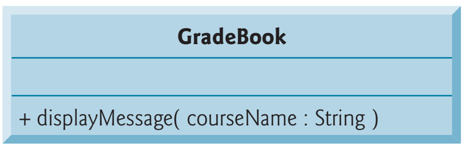


Fig. 3.6 | UML class diagram indicating that class `GradeBook` has a `displayMessage` operation with a `courseName` parameter of UML type `String`.

Notes on `import` *Declarations*

Notice the `import` declaration in Fig. 3.5 (line 4). This indicates to the compiler that the program uses class `Scanner`. Why do we need to import class `Scanner`, but not classes `System`, `String` or `GradeBook`? Classes `System` and `String` are in package `java.lang`, which is implicitly imported into *every* Java program, so all programs can use that package's classes *without* explicitly importing them. Most other classes you'll use in Java programs must be imported explicitly.

There's a special relationship between classes that are compiled in the same directory on disk, like classes `GradeBook` and `GradeBookTest`. By default, such classes are considered to be in the same package—known as the **default package**. Classes in the same package are *implicitly imported* into the source-code files of other classes in the same package. Thus, an `import` declaration is *not* required when one class in a package uses another in the same package—such as when class `GradeBookTest` uses class `GradeBook`.

The `import` declaration in line 4 is *not* required if we always refer to class `Scanner` as `java.util.Scanner`, which includes the *full package name and class name*. This is known as the class's **fully qualified class name**. For example, line 12 could be written as

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

Software Engineering Observation 3.1

The Java compiler does not require `import` *declarations in a Java source-code file if the fully qualified class name is specified every time a class name is used in the source code. Most Java programmers prefer to use* `import` *declarations.*

3.4 Instance Variables, *set* Methods and *get* Methods

In Chapter 2, we declared all of an application's variables in the application's `main` method. Variables declared in the body of a particular method are known as **local variables** and can be used only in that method. When that method terminates, the values of its local variables are lost. Recall from Section 1.6 that an object has *attributes* that are carried with it as it's used in a program. Such attributes exist before a method is called on an object, while the method is executing and after the method completes execution.

A class normally consists of one or more methods that manipulate the attributes that belong to a particular object of the class. Attributes are represented as variables in a class declaration. Such variables are called **fields** and are declared *inside* a class declaration but *outside* the bodies of the class's method declarations. When each object of a class maintains its own copy of an attribute, the field that represents the attribute is also known as an **instance variable**—each object (instance) of the class has a separate instance of the variable in memory. The example in this section demonstrates a `GradeBook` class that contains a `courseName` instance variable to represent a particular `GradeBook` object's course name.

`GradeBook` *Class with an Instance Variable, a* set *Method and a* get *Method*

In our next application (Figs. 3.7–3.8), class `GradeBook` (Fig. 3.7) maintains the course name as an instance variable so that it can be used or modified at any time during an application's execution. The class contains three methods—`setCourseName`, `getCourseName` and `displayMessage`. Method `setCourseName` stores a course name in a `GradeBook`. Method `getCourseName` obtains a `GradeBook`'s course name. Method `displayMessage`, which now specifies no parameters, still displays a welcome message that includes the course name; as you'll see, the method now obtains the course name by calling a method in the same class—`getCourseName`.

```
1  // Fig. 3.7: GradeBook.java
2  // GradeBook class that contains a courseName instance variable
3  // and methods to set and get its value.
4
5  public class GradeBook
6  {
7     private String courseName; // course name for this GradeBook
8
9     // method to set the course name
10    public void setCourseName( String name )
11    {
12       courseName = name; // store the course name
13    } // end method setCourseName
14
15    // method to retrieve the course name
16    public String getCourseName()
17    {
18       return courseName;
19    } // end method getCourseName
20
21    // display a welcome message to the GradeBook user
22    public void displayMessage()
23    {
24       // calls getCourseName to get the name of
25       // the course this GradeBook represents
26       System.out.printf( "Welcome to the grade book for\n%s!\n",
27          getCourseName() );
28    } // end method displayMessage
29 } // end class GradeBook
```

Fig. 3.7 | `GradeBook` class that contains a `courseName` instance variable and methods to set and get its value.

A typical instructor teaches more than one course, each with its own course name. Line 7 declares `courseName` as a variable of type `String`. Because the variable is declared *in* the body of the class but *outside* the bodies of the class's methods (lines 10–13, 16–19 and 22–28), line 7 is a declaration for an *instance variable*. Every instance (i.e., object) of class `GradeBook` contains one copy of each instance variable. For example, if there are two `GradeBook` objects, each object has its own copy of `courseName`. A benefit of making `courseName` an instance variable is that all the methods of the class (in this case, `GradeBook`) can manipulate any instance variables that appear in the class (in this case, `courseName`).

Access Modifiers `public` and `private`

Most instance-variable declarations are preceded with the keyword `private` (as in line 7). Like `public`, keyword **`private`** is an *access modifier*. *Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they're declared.* Thus, variable `courseName` can be used only in methods `setCourseName`, `getCourseName` and `displayMessage` of (every object of) class `GradeBook`.

Declaring instance variables with access modifier `private` is known as **data hiding** or information hiding. When a program creates (instantiates) an object of class `GradeBook`, variable `courseName` is *encapsulated* (hidden) in the object and can be accessed only by methods of the object's class. This prevents `courseName` from being modified accidentally by a class in another part of the program. In class `GradeBook`, methods `setCourseName` and `getCourseName` manipulate the instance variable `courseName`.

Software Engineering Observation 3.2

Precede each field and method declaration with an access modifier. Generally, instance variables should be declared `private` and methods `public`. (It's appropriate to declare certain methods `private`, if they'll be accessed only by other methods of the class.)

Good Programming Practice 3.1

We prefer to list a class's fields first, so that, as you read the code, you see the names and types of the variables before they're used in the class's methods. You can list the class's fields anywhere in the class outside its method declarations, but scattering them can lead to hard-to-read code.

Methods `setCourseName` and `getCourseName`

Method `setCourseName` (lines 10–13) does not return any data when it completes its task, so its return type is `void`. The method receives one parameter—`name`—which represents the course name that will be passed to the method as an argument. Line 12 assigns `name` to instance variable `courseName`.

Method `getCourseName` (lines 16–19) returns a particular `GradeBook` object's `courseName`. The method has an empty parameter list, so it does not require additional information to perform its task. The method specifies that it returns a `String`—this is the method's return type. When a method that specifies a return type other than `void` is called and completes its task, the method returns a *result* to its calling method. For example, when you go to an automated teller machine (ATM) and request your account balance, you expect the ATM to give you back a value that represents your balance. Similarly, when a statement calls method `getCourseName` on a `GradeBook` object, the statement expects to receive the `GradeBook`'s course name (in this case, a `String`, as specified in the method declaration's return type).

The **`return`** statement in line 18 passes the value of instance variable `courseName` back to the statement that calls method `getCourseName`. Consider, method `displayMessage`'s line 27, which calls method `getCourseName`. When the value is returned, the statement in lines 26–27 uses that value to output the course name. Similarly, if you have a method `square` that returns the square of its argument, you'd expect the statement

```
int result = square( 2 );
```

to return `4` from method `square` and assign `4` to the variable `result`. If you have a method `maximum` that returns the largest of three integer arguments, you'd expect the statement

```
int biggest = maximum( 27, 114, 51 );
```

to return `114` from method `maximum` and assign 114 to variable `biggest`.

The statements in lines 12 and 18 each use `courseName` *even though it was not declared in any of the methods*. We can use `courseName` in `GradeBook`'s methods because `courseName` is an instance variable of the class.

Method `displayMessage`

Method `displayMessage` (lines 22–28) does *not* return any data when it completes its task, so its return type is `void`. The method does *not* receive parameters, so the parameter list is empty. Lines 26–27 output a welcome message that includes the value of instance variable `courseName`, which is returned by the call to method `getCourseName` in line 27. Notice that one method of a class (`displayMessage` in this case) can call another method of the *same* class by using just the method name (`getCourseName` in this case).

`GradeBookTest` *Class That Demonstrates Class* `GradeBook`

Class `GradeBookTest` (Fig. 3.8) creates one object of class `GradeBook` and demonstrates its methods. Line 14 creates a `GradeBook` object and assigns it to local variable `myGradeBook` of type `GradeBook`. Lines 17–18 display the initial course name calling the object's `getCourseName` method. The first line of the output shows the name "`null`." *Unlike local variables, which are not automatically initialized, every field has a* ***default initial value****—a value provided by Java when you do not specify the field's initial value.* Thus, fields are *not* required to be explicitly initialized before they're used in a program—unless they must be initialized to values *other than* their default values. The default value for a field of type `String` (like `courseName` in this example) is `null`, which we say more about in Section 3.5.

Line 21 prompts the user to enter a course name. Local `String` variable `theName` (declared in line 22) is initialized with the course name entered by the user, which is returned by the call to the `nextLine` method of the `Scanner` object `input`. Line 23 calls object `myGradeBook`'s `setCourseName` method and supplies `theName` as the method's argument. When the method is called, the argument's value is assigned to parameter `name` (line 10, Fig. 3.7) of method `setCourseName` (lines 10–13, Fig. 3.7). Then the parameter's value is assigned to instance variable `courseName` (line 12, Fig. 3.7). Line 24 (Fig. 3.8) skips a line in the output, then line 27 calls object `myGradeBook`'s `displayMessage` method to display the welcome message containing the course name.

```
1  // Fig. 3.8: GradeBookTest.java
2  // Creating and manipulating a GradeBook object.
3  import java.util.Scanner; // program uses Scanner
4
5  public class GradeBookTest
6  {
7     // main method begins program execution
8     public static void main( String[] args )
9     {
10       // create Scanner to obtain input from command window
11       Scanner input = new Scanner( System.in );
12
13       // create a GradeBook object and assign it to myGradeBook
14       GradeBook myGradeBook = new GradeBook();
15
```

Fig. 3.8 | Creating and manipulating a `GradeBook` object. (Part 1 of 2.)

```
        // display initial value of courseName
        System.out.printf( "Initial course name is: %s\n\n",
           myGradeBook.getCourseName() );

        // prompt for and read course name
        System.out.println( "Please enter the course name:" );
        String theName = input.nextLine(); // read a line of text
        myGradeBook.setCourseName( theName ); // set the course name
        System.out.println(); // outputs a blank line

        // display welcome message after specifying course name
        myGradeBook.displayMessage();
     } // end main
  } // end class GradeBookTest
```

```
Initial course name is: null

Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

Fig. 3.8 | Creating and manipulating a `GradeBook` object. (Part 2 of 2.)

set *and* get *Methods*

A class's `private` fields can be manipulated *only* by the class's methods. So a **client of an object**—that is, any class that calls the object's methods—calls the class's `public` methods to manipulate the `private` fields of an object of the class. This is why the statements in method `main` (Fig. 3.8) call the `setCourseName`, `getCourseName` and `displayMessage` methods on a `GradeBook` object. Classes often provide `public` methods to allow clients to ***set*** (i.e., assign values to) or ***get*** (i.e., obtain the values of) `private` instance variables. The names of these methods need not begin with *set* or *get*, but this naming convention is recommended and is convention for special Java software components called JavaBeans, which can simplify programming in many Java integrated development environments (IDEs). The method that *sets* instance variable `courseName` in this example is called `setCourseName`, and the method that *gets* its value is called `getCourseName`.

`GradeBook` UML Class Diagram with an Instance Variable and set *and* get *Methods*

Figure 3.9 contains an updated UML class diagram for the version of class `GradeBook` in Fig. 3.7. This diagram models class `GradeBook`'s instance variable `courseName` as an attribute in the middle compartment of the class. The UML represents instance variables as attributes by listing the attribute name, followed by a colon and the attribute type. The UML type of attribute `courseName` is `String`. Instance variable `courseName` is `private` in Java, so the class diagram lists a minus sign (–) access modifier in front of the corresponding attribute's name. Class `GradeBook` contains three `public` methods, so the class diagram lists three operations in the third compartment. Recall that the plus sign (+) before each operation name indicates that the operation is `public`. Operation `setCourseName` has a `String` parameter called `name`. The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name. Method `getCourseName` of class `GradeBook` (Fig. 3.7) has a `String` return type in Java, so the class diagram shows a `String` return type in the UML. Operations `setCourseName` and `displayMessage` *do not* return values (i.e., they return `void` in Java), so the UML class diagram *does not* specify a return type after the parentheses of these operations.

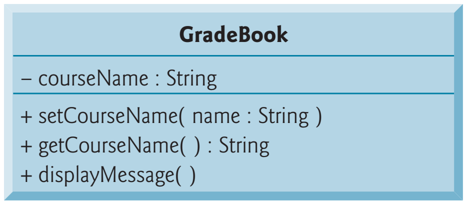


Fig. 3.9 | UML class diagram indicating that class `GradeBook` has a private `courseName` attribute of UML type `String` and three public operations—`setCourseName` (with a `name` parameter of UML type `String`), `getCourseName` (which returns UML type `String`) and `displayMessage`.

3.5 Primitive Types vs. Reference Types

Java's types are divided into primitive types and **reference types**. The primitive types are `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`. All nonprimitive types are reference types, so classes, which specify the types of objects, are reference types.

A primitive-type variable can store exactly one *value of its declared type* at a time. For example, an `int` variable can store one whole number (such as 7) at a time. When another value is assigned to that variable, its initial value is replaced. Primitive-type instance variables are *initialized by default*—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

```
private int numberOfStudents = 10;
```

Recall that local variables are *not* initialized by default.

Error-Prevention Tip 3.1

An attempt to use an uninitialized local variable causes a compilation error.

Programs use variables of reference types (normally called **references**) to store the *locations* 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. Line 14 of Fig. 3.8 creates an object of class `GradeBook`, and the variable `myGradeBook` contains a reference to that `GradeBook` object. *Reference-type instance variables are initialized by default to the value* `null`—a reserved word that represents a "reference to nothing." This is why the first call to `getCourseName` in line 18 of Fig. 3.8 returned `null`—the value of `courseName` had not been set, so the default initial value `null` was returned. The complete list of reserved words and keywords is listed in Appendix C.

When you use an object of another class, a reference to the object is required to **invoke** (i.e., call) its methods. In the application of Fig. 3.8, the statements in method `main` use the variable `myGradeBook` to send messages to the `GradeBook` object. These messages are calls to methods (like `setCourseName` and `getCourseName`) that enable the program to interact with the `GradeBook` object. For example, the statement in line 23 uses `myGradeBook` to send the `setCourseName` message to the `GradeBook` object. The message includes the argument that `setCourseName` requires to perform its task. The `GradeBook` object uses this information to set the `courseName` instance variable. Primitive-type variables do not refer to objects, so such variables cannot be used to invoke methods.

Software Engineering Observation 3.3

A variable's declared type (e.g., `int`*,* `double` *or* `GradeBook`*) indicates whether the variable is of a primitive or a reference type. If a variable is* not *of one of the eight primitive types, then it's of a reference type.*

3.6 Initializing Objects with Constructors

As mentioned in Section 3.4, when an object of class `GradeBook` (Fig. 3.7) is created, its instance variable `courseName` is initialized to `null` by default. What if you want to provide a course name when you create a `GradeBook` object? Each class you declare can provide a special method called a constructor that can be used to initialize an object of a class when the object is created. In fact, Java *requires* a constructor call for *every* object that's created. Keyword `new` requests memory from the system to store an object, then calls the corresponding class's constructor to initialize the object. The call is indicated by the parentheses after the class name. A constructor *must* have the *same name* as the class. For example, line 14 of Fig. 3.8 first uses `new` to create a `GradeBook` object. The empty parentheses after "`new GradeBook`" indicate a call to the class's constructor without arguments. By default, the compiler provides a **default constructor** with *no parameters* in any class that does *not* explicitly include a constructor. When a class has only the default constructor, its instance variables are initialized to their *default values*.

When you declare a class, you can provide your own constructor to specify custom initialization for objects of your class. For example, you might want to specify a course name for a `GradeBook` object when the object is created, as in

```
GradeBook myGradeBook =
   new GradeBook( "CS101 Introduction to Java Programming" );
```

In this case, the argument `"CS101 Introduction to Java Programming"` is passed to the `GradeBook` object's constructor and used to initialize the `courseName`. The preceding statement requires that the class provide a constructor with a `String` parameter. Figure 3.10 contains a modified `GradeBook` class with such a constructor.

```
// Fig. 3.10: GradeBook.java
// GradeBook class with a constructor to initialize the course name.

public class GradeBook
{
   private String courseName; // course name for this GradeBook

```

Fig. 3.10 | `GradeBook` class with a constructor to initialize the course name. (Part 1 of 2.)

```java
8     // constructor initializes courseName with String argument
9     public GradeBook( String name ) // constructor name is class name
10    {
11       courseName = name; // initializes courseName
12    } // end constructor
13
14    // method to set the course name
15    public void setCourseName( String name )
16    {
17       courseName = name; // store the course name
18    } // end method setCourseName
19
20    // method to retrieve the course name
21    public String getCourseName()
22    {
23       return courseName;
24    } // end method getCourseName
25
26    // display a welcome message to the GradeBook user
27    public void displayMessage()
28    {
29       // this statement calls getCourseName to get the
30       // name of the course this GradeBook represents
31       System.out.printf( "Welcome to the grade book for\n%s!\n",
32          getCourseName() );
33    } // end method displayMessage
34 } // end class GradeBook
```

Fig. 3.10 | `GradeBook` class with a constructor to initialize the course name. (Part 2 of 2.)

Lines 9–12 declare `GradeBook`'s constructor. Like a method, a constructor's parameter list specifies the data it requires to perform its task. When you create a new object (as we'll do in Fig. 3.11), this data is placed in the *parentheses that follow the class name*. Line 9 of Fig. 3.10 indicates that the constructor has a `String` parameter called `name`. The `name` passed to the constructor is assigned to instance variable `courseName` in line 11.

Figure 3.11 initializes `GradeBook` objects using the constructor. Lines 11–12 create and initialize the `GradeBook` object `gradeBook1`. The `GradeBook` constructor is called with the argument `"CS101 Introduction to Java Programming"` to initialize the course name. The class instance creation expression in lines 11–12 returns a reference to the new object, which is assigned to the variable `gradeBook1`. Lines 13–14 repeat this process, this time passing the argument `"CS102 Data Structures in Java"` to initialize the course name for `gradeBook2`. Lines 17–20 use each object's `getCourseName` method to obtain the course names and show that they were initialized when the objects were created. The output confirms that each `GradeBook` maintains its own copy of instance variable `courseName`.

An important difference between constructors and methods is that constructors cannot return values, so they cannot specify a return type (not even `void`). Normally, constructors are declared `public`. If a class does not include a constructor, the class's instance variables are initialized to their default values. *If you declare any constructors for a class, the Java compiler will not create a default constructor for that class.* Thus, we can no longer create a `GradeBook` object with `new GradeBook()` as we did in the earlier examples.

```java
// Fig. 3.11: GradeBookTest.java
// GradeBook constructor used to specify the course name at the
// time each GradeBook object is created.

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String[] args )
   {
      // create GradeBook object
      GradeBook gradeBook1 = new GradeBook(
         "CS101 Introduction to Java Programming" );
      GradeBook gradeBook2 = new GradeBook(
         "CS102 Data Structures in Java" );

      // display initial value of courseName for each GradeBook
      System.out.printf( "gradeBook1 course name is: %s\n",
         gradeBook1.getCourseName() );
      System.out.printf( "gradeBook2 course name is: %s\n",
         gradeBook2.getCourseName() );
   } // end main
} // end class GradeBookTest
```

```
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
```

Fig. 3.11 | `GradeBook` constructor used to specify the course name at the time each `GradeBook` object is created.

Software Engineering Observation 3.4

Unless default initialization of your class's instance variables is acceptable, provide a constructor to ensure that they're properly initialized with meaningful values when each new object of your class is created.

Adding the Constructor to Class `GradeBook`*'s UML Class Diagram*

The UML class diagram of Fig. 3.12 models class `GradeBook` of Fig. 3.10, which has a constructor that has a `name` parameter of type `String`. Like operations, the UML models constructors in the third compartment of a class in a class diagram. To distinguish a

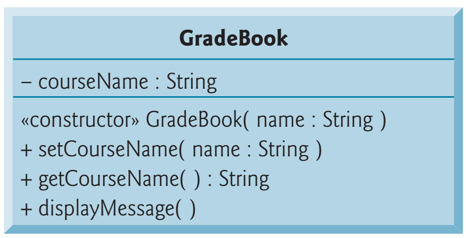


Fig. 3.12 | UML class diagram indicating that class `GradeBook` has a constructor that has a `name` parameter of UML type `String`.

constructor from a class's operations, the UML requires that the word "constructor" be placed between **guillemets (« and »)** before the constructor's name. It's *customary* to list constructors *before* other operations in the third compartment.

Constructors with Multiple Parameters

Sometimes you'll want to initialize objects with multiple data items. In Exercise 3.11, we ask you to store the course name *and* the instructor's name in a `GradeBook` object. In this case, the `GradeBook`'s constructor would be modified to receive two `String`s, as in

```
public GradeBook( String courseName, String instructorName )
```

and you'd call the `GradeBook` constructor as follows:

```
GradeBook gradeBook = new GradeBook(
   "CS101 Introduction to Java Programming", "Sue Green" );
```

3.7 Floating-Point Numbers and Type `double`

We now depart temporarily from our `GradeBook` case study to declare an `Account` class that maintains the balance of a bank account. Most account balances are not whole numbers (such as 0, –22 and 1024). For this reason, class `Account` represents the account balance as a **floating-point number** (i.e., a number with a decimal point, such as 7.33, 0.0975 or 1000.12345). Java provides two primitive types for storing floating-point numbers in memory—`float` and `double`. They differ primarily in that `double` variables can store numbers with larger magnitude and finer detail (i.e., more digits to the right of the decimal point—also known as the number's **precision**) than `float` variables.

Floating-Point Number Precision and Memory Requirements

Variables of type **`float`** represent **single-precision floating-point numbers** and can represent up to *seven significant digits*. Variables of type **`double`** represent **double-precision floating-point numbers**. These require twice as much memory as `float` variables and provide *15 significant digits*—approximately double the precision of `float` variables. For the range of values required by most programs, variables of type `float` should suffice, but you can use `double` to "play it safe." In some applications, even `double` variables will be inadequate. Most programmers represent floating-point numbers with type `double`. In fact, Java treats all floating-point numbers you type in a program's source code (such as 7.33 and 0.0975) as `double` values by default. Such values in the source code are known as **floating-point literals**. See Appendix D, Primitive Types, for the ranges of values for `float`s and `double`s.

Although floating-point numbers are not always 100% precise, they have numerous applications. For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it may actually be 98.5999473210643. Calling this number simply 98.6 is fine for most applications involving body temperatures. Owing to the imprecise nature of floating-point numbers, type `double` is preferred over type `float`, because `double` variables can represent floating-point numbers more accurately. For this reason, we primarily use type `double` throughout the book. For precise floating-point numbers, Java provides class `BigDecimal` (package `java.math`).

Floating-point numbers also arise as a result of division. In conventional arithmetic, when we divide 10 by 3, the result is 3.3333333…, with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so clearly the stored floating-point value can be only an approximation.

`Account` Class with an Instance Variable of Type `double`

Our next application (Figs. 3.13–3.14) contains a class named `Account` (Fig. 3.13) that maintains the balance of a bank account. A typical bank services many accounts, each with its own balance, so line 7 declares an instance variable named `balance` of type `double`. It's an instance variable because it's declared in the body of the class but outside the class's method declarations (lines 10–16, 19–22 and 25–28). Every instance (i.e., object) of class `Account` contains its own copy of `balance`.

```
1  // Fig. 3.13: Account.java
2  // Account class with a constructor to validate and
3  // initialize instance variable balance of type double.
4
5  public class Account
6  {
7     private double balance; // instance variable that stores the balance
8
9     // constructor
10    public Account( double initialBalance )
11    {
12       // validate that initialBalance is greater than 0.0;
13       // if it is not, balance is initialized to the default value 0.0
14       if ( initialBalance > 0.0 )
15          balance = initialBalance;
16    } // end Account constructor
17
18    // credit (add) an amount to the account
19    public void credit( double amount )
20    {
21       balance = balance + amount; // add amount to balance
22    } // end method credit
23
24    // return the account balance
25    public double getBalance()
26    {
27       return balance; // gives the value of balance to the calling method
28    } // end method getBalance
29 } // end class Account
```

Fig. 3.13 | `Account` class with a constructor to validate and initialize instance variable `balance` of type `double`.

The class has a constructor and two methods. It's common for someone opening an account to deposit money immediately, so the constructor (lines 10–16) receives a parameter `initialBalance` of type `double` that represents the *starting balance*. Lines 14–15 ensure that `initialBalance` is greater than `0.0`. If so, `initialBalance`'s value is assigned to instance variable `balance`. Otherwise, `balance` remains at `0.0`—its default initial value.

Method `credit` (lines 19–22) does *not* return any data when it completes its task, so its return type is `void`. The method receives one parameter named `amount`—a `double` value that will be added to the balance. Line 21 adds `amount` to the current value of `balance`, then assigns the result to `balance` (thus replacing the prior balance amount).

Method `getBalance` (lines 25–28) allows clients of the class (i.e., other classes that use this class) to obtain the value of a particular `Account` object's `balance`. The method specifies return type `double` and an empty parameter list.

Once again, the statements in lines 15, 21 and 27 use instance variable `balance` even though it was *not* declared in any of the methods. We can use `balance` in these methods because it's an instance variable of the class.

`AccountTest` Class to Use Class `Account`

Class `AccountTest` (Fig. 3.14) creates two `Account` objects (lines 10–11) and initializes them with `50.00` and `-7.53`, respectively. Lines 14–17 output the balance in each `Account` by calling the `Account`'s `getBalance` method. When method `getBalance` is called for `account1` from line 15, the value of `account1`'s balance is returned from line 27 of Fig. 3.13 and displayed by the `System.out.printf` statement (Fig. 3.14, lines 14–15). Similarly, when method `getBalance` is called for `account2` from line 17, the value of the `account2`'s balance is returned from line 27 of Fig. 3.13 and displayed by the `System.out.printf` statement (Fig. 3.14, lines 16–17). The balance of `account2` is `0.00`, because the constructor ensured that the account could *not* begin with a negative balance. The value is output by `printf` with the format specifier `%.2f`. The format specifier **`%f`** is used to output values of type `float` or `double`. The `.2` between `%` and `f` represents the number of decimal places (2) that should be output to the right of the decimal point in the floating-point number—also known as the number's **precision**. Any floating-point value output with `%.2f` will be rounded to the hundredths position—for example, 123.457 would be rounded to 123.46, 27.333 would be rounded to 27.33 and 123.455 would be rounded to 123.46.

```
1  // Fig. 3.14: AccountTest.java
2  // Inputting and outputting floating-point numbers with Account objects.
3  import java.util.Scanner;
4
5  public class AccountTest
6  {
7     // main method begins execution of Java application
8     public static void main( String[] args )
9     {
10       Account account1 = new Account( 50.00 ); // create Account object
11       Account account2 = new Account( -7.53 ); // create Account object
12
13       // display initial balance of each object
14       System.out.printf( "account1 balance: $%.2f\n",
15          account1.getBalance() );
16       System.out.printf( "account2 balance: $%.2f\n\n",
17          account2.getBalance() );
18
19       // create Scanner to obtain input from command window
20       Scanner input = new Scanner( System.in );
21       double depositAmount; // deposit amount read from user
```

Fig. 3.14 | Inputting and outputting floating-point numbers with `Account` objects. (Part 1 of 2.)

```
22
23         System.out.print( "Enter deposit amount for account1: " ); // prompt
24         depositAmount = input.nextDouble(); // obtain user input
25         System.out.printf( "\nadding %.2f to account1 balance\n\n",
26            depositAmount );
27         account1.credit( depositAmount ); // add to account1 balance
28
29         // display balances
30         System.out.printf( "account1 balance: $%.2f\n",
31            account1.getBalance() );
32         System.out.printf( "account2 balance: $%.2f\n\n",
33            account2.getBalance() );
34
35         System.out.print( "Enter deposit amount for account2: " ); // prompt
36         depositAmount = input.nextDouble(); // obtain user input
37         System.out.printf( "\nadding %.2f to account2 balance\n\n",
38            depositAmount );
39         account2.credit( depositAmount ); // add to account2 balance
40
41         // display balances
42         System.out.printf( "account1 balance: $%.2f\n",
43            account1.getBalance() );
44         System.out.printf( "account2 balance: $%.2f\n",
45            account2.getBalance() );
46      } // end main
47   } // end class AccountTest
```

```
account1 balance: $50.00
account2 balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

account1 balance: $75.53
account2 balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

account1 balance: $75.53
account2 balance: $123.45
```

Fig. 3.14 | Inputting and outputting floating-point numbers with `Account` objects. (Part 2 of 2.)

Line 21 declares local variable `depositAmount` to store each deposit amount entered by the user. Unlike the instance variable `balance` in class `Account`, local variable `depositAmount` in `main` is *not* initialized to 0.0 by default. However, this variable does not need to be initialized here, because its value will be determined by the user's input.

Line 23 prompts the user to enter a deposit amount for `account1`. Line 24 obtains the input from the user by calling `Scanner` object `input`'s **`nextDouble`** method, which returns a `double` value entered by the user. Lines 25–26 display the deposit amount. Line 27 calls object account1's credit method and supplies depositAmount as the method's argument. When the method is called, the argument's value is assigned to parameter amount (line 19 of Fig. 3.13) of method credit (lines 19–22 of Fig. 3.13); then method credit adds that value to the balance (line 21 of Fig. 3.13). Lines 30–33 (Fig. 3.14) output the balances of both Accounts again to show that only account1's balance changed.

Line 35 prompts the user to enter a deposit amount for account2. Line 36 obtains the input from the user by calling Scanner object input's nextDouble method. Lines 37–38 display the deposit amount. Line 39 calls object account2's credit method and supplies depositAmount as the method's argument; then method credit adds that value to the balance. Finally, lines 42–45 output the balances of both Accounts again to show that only account2's balance changed.

UML Class Diagram for Class `Account`

The UML class diagram in Fig. 3.15 models class Account of Fig. 3.13. The diagram models the private attribute balance with UML type Double to correspond to the class's instance variable balance of Java type double. The diagram models class Account's constructor with a parameter initialBalance of UML type Double in the third compartment of the class. The class's two public methods are modeled as operations in the third compartment as well. The diagram models operation credit with an amount parameter of UML type Double (because the corresponding method has an amount parameter of Java type double), and operation getBalance with a return type of Double (because the corresponding Java method returns a double value).

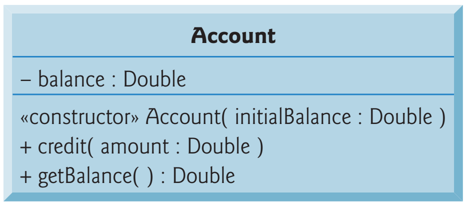


Fig. 3.15 | UML class diagram indicating that class Account has a private balance attribute of UML type Double, a constructor (with a parameter of UML type Double) and two public operations—credit (with an amount parameter of UML type Double) and getBalance (returns UML type Double).

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

This optional case study is designed for those who want to begin learning Java's powerful capabilities for creating graphical user interfaces (GUIs) and graphics early in the book, before the main discussions of these topics in Chapter 14, GUI Components: Part 1, Chapter 15, Graphics and Java 2D, and Chapter 25, GUI Components: Part 2.

The GUI and Graphics Case Study appears in 10 brief sections (see Fig. 3.16). Each section introduces new concepts and provides examples with screen captures that show sample interactions and results. In the first few sections, you'll create your first graphical applications. In subsequent sections, you'll use object-oriented programming concepts to create an application that draws a variety of shapes. When we formally introduce GUIs in Chapter 14, we use the mouse to choose exactly which shapes to draw and where to draw them. In Chapter 15, we add capabilities of the Java 2D graphics API to draw the shapes with different line thicknesses and fills. We hope you find this case study informative and entertaining.

Location	Title—Exercise(s)
Section 3.8	Using Dialog Boxes—Basic input and output with dialog boxes
Section 4.14	Creating Simple Drawings—Displaying and drawing lines on the screen
Section 5.10	Drawing Rectangles and Ovals—Using shapes to represent data
Section 6.13	Colors and Filled Shapes—Drawing a bull's-eye and random graphics
Section 7.15	Drawing Arcs—Drawing spirals with arcs
Section 8.16	Using Objects with Graphics—Storing shapes as objects
Section 9.8	Displaying Text and Images Using Labels—Providing status information
Section 10.8	Drawing with Polymorphism—Identifying the similarities between shapes
Exercise 14.17	Expanding the Interface—Using GUI components and event handling
Exercise 15.31	Adding Java 2D—Using the Java 2D API to enhance drawings

Fig. 3.16 | Summary of the GUI and Graphics Case Study in each chapter.

Displaying Text in a Dialog Box

The programs presented thus far display output in the command window. Many applications use windows or **dialog boxes** (also called **dialogs**) to display output. Web browsers such as Firefox, Internet Explorer, Chrome and Safari display web pages in their own windows. E-mail programs allow you to type and read messages in a window. Typically, dialog boxes are windows in which programs display important messages to users. Class **`JOptionPane`** provides prebuilt dialog boxes that enable programs to display windows containing messages—such windows are called **message dialogs**. Figure 3.17 displays the `String` `"Welcome\nto\nJava"` in a message dialog.

```java
// Fig. 3.17: Dialog1.java
// Using JOptionPane to display multiple lines in a dialog box.
import javax.swing.JOptionPane; // import class JOptionPane

public class Dialog1
{
   public static void main( String[] args )
   {
      // display a dialog with a message
      JOptionPane.showMessageDialog( null, "Welcome\nto\nJava" );
   } // end main
} // end class Dialog1
```

Fig. 3.17 | Using `JOptionPane` to display multiple lines in a dialog box. (Part 1 of 2.)

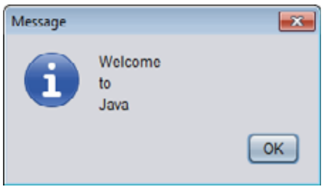


Fig. 3.17 | Using `JOptionPane` to display multiple lines in a dialog box. (Part 2 of 2.)

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

Introducing `static` *Methods*

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

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

Notice that you do *not* create an object of class `JOptionPane` to use its `static` method `showMessageDialog`. We discuss `static` methods in more detail in Chapter 6.

Entering Text in a Dialog

Figure 3.18 uses another predefined `JOptionPane` dialog called an **input dialog** that allows the user to enter data into a program. The program asks for the user's name and responds with a message dialog containing a greeting and the name that the user entered.

Lines 10–11 use `JOptionPane` method **`showInputDialog`** to display an input dialog containing a prompt and a field (known as a **text field**) in which the user can enter text. Method `showInputDialog`'s argument is the prompt that indicates what the user should enter. The user types characters in the text field, then clicks the **OK** button or presses the *Enter* key to return the `String` to the program. Method `showInputDialog` (line 11) returns a `String` containing the characters typed by the user. We store the `String` in variable `name` (line 10). [*Note:* If you press the dialog's **Cancel** button or press the *Esc* key, the method returns `null` and the program displays the word "null" as the name.]

Lines 14–15 use `static String` method **`format`** to return a `String` containing a greeting with the user's name. Method `format` works like method `System.out.printf`, except that `format` returns the formatted `String` rather than displaying it in a command window. Line 18 displays the greeting in a message dialog, just as we did in Fig. 3.17.

```java
// Fig. 3.18: NameDialog.java
// Basic input with a dialog box.
import javax.swing.JOptionPane;

public class NameDialog
{
   public static void main( String[] args )
   {
      // prompt user to enter name
      String name =
         JOptionPane.showInputDialog( "What is your name?" );

      // create the message
      String message =
         String.format( "Welcome, %s, to Java Programming!", name );

      // display the message to welcome the user by name
      JOptionPane.showMessageDialog( null, message );
   } // end main
} // end class NameDialog
```

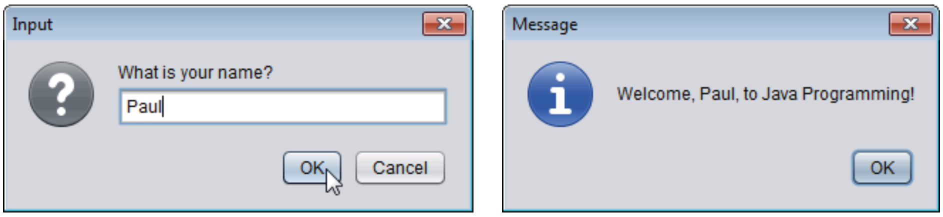


Fig. 3.18 | Obtaining user input from a dialog.

GUI and Graphics Case Study Exercise

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

3.9 Wrap-Up

In this chapter, you learned how to declare instance variables of a class to maintain data for each object of the class, and how to declare methods that operate on that data. You learned how to call a method to tell it to perform its task and how to pass information to methods as arguments. You learned the difference between a local variable of a method and an instance variable of a class and that only instance variables are initialized automatically. You also learned how to use a class's constructor to specify the initial values for an object's instance variables. Throughout the chapter, you saw how the UML can be used to create class diagrams that model the constructors, methods and attributes of classes. Finally, you learned about floating-point numbers—how to store them with variables of primitive type `double`, how to input them with a `Scanner` object and how to format them with `printf` and format specifier `%f` for display purposes. In the next chapter we begin our introduction to control statements, which specify the order in which a program's actions are performed. You'll use these in your methods to specify how they should perform their tasks.

Summary

Section 3.2 Declaring a Class with a Method and Instantiating an Object of a Class

- Each class declaration that begins with the access modifier (p. 72) `public` must be stored in a file that has exactly the same name as the class and ends with the `.java` file-name extension.
- Every class declaration contains keyword `class` followed immediately by the class's name.
- A method declaration that begins with keyword `public` indicates that the method can be called by other classes declared outside the class declaration.
- Keyword `void` indicates that a method will perform a task but will not return any information.
- By convention, method names begin with a lowercase first letter and all subsequent words in the name begin with a capital first letter.
- Empty parentheses following a method name indicate that the method does not require any parameters to perform its task.
- Every method's body is delimited by left and right braces (`{` and `}`).
- The method's body contains statements that perform the method's task. After the statements execute, the method has completed its task.
- When you attempt to execute a class, Java looks for the class's `main` method to begin execution.
- Typically, you cannot call a method of another class until you create an object of that class.
- A class instance creation expression (p. 74) begins with keyword `new` and creates a new object.
- To call a method of an object, follow the variable name with a dot separator (`.`; p. 75), the method name and a set of parentheses containing the method's arguments.
- In the UML, each class is modeled in a class diagram as a rectangle with three compartments. The top compartment contains the class's name centered horizontally in boldface. The middle one contains the class's attributes, which correspond to fields (p. 79) in Java. The bottom one contains the class's operations (p. 76), which correspond to methods and constructors in Java.
- The UML models operations by listing the operation name followed by a set of parentheses. A plus sign (+) in front of the operation name indicates that the operation is a `public` one in the UML (i.e., a `public` method in Java).

Section 3.3 Declaring a Method with a Parameter

- Methods often require parameters (p. 76) to perform their tasks. Such additional information is provided to methods via arguments in method calls.
- `Scanner` method `nextLine` (p. 76) reads characters until a newline character is encountered, then returns the characters as a `String`.
- `Scanner` method `next` (p. 77) reads characters until any white-space character is encountered, then returns the characters as a `String`.
- A method that requires data to perform its task must specify this in its declaration by placing additional information in the method's parameter list (p. 76).
- Each parameter must specify both a type and a variable name.

- At the time a method is called, its arguments are assigned to its parameters. Then the method body uses the parameter variables to access the argument values.
- A method specifies multiple parameters in a comma-separated list.
- The number of arguments in the method call must match the number of parameters in the method declaration's parameter list. Also, the argument types in the method call must be consistent with the types of the corresponding parameters in the method's declaration.
- Class `String` is in package `java.lang`, which is imported implicitly into all source-code files.
- By default, classes compiled into the same directory are in the same package. Classes in the same package are implicitly imported into the source-code files of other classes in the same package.
- `import` declarations are not required if you always use fully qualified class names (p. 79).
- The UML models a parameter of an operation by listing the parameter name, followed by a colon and the parameter type between the parentheses following the operation name.
- The UML has its own data types similar to those of Java. Not all the UML data types have the same names as the corresponding Java types.
- The UML type `String` corresponds to the Java type `String`.

Section 3.4 Instance Variables, set *Methods and* get *Methods*

- Variables declared in a method's body are local variables and can be used only in that method.
- A class normally consists of one or more methods that manipulate the attributes (data) that belong to a particular object of the class. Such variables are called fields and are declared inside a class declaration but outside the bodies of the class's method declarations.
- When each object of a class maintains its own copy of an attribute, the corresponding field is known as an instance variable.
- Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they're declared.
- Declaring instance variables with access modifier `private` (p. 80) is known as data hiding.
- A benefit of fields is that all the methods of the class can use the fields. Another distinction between a field and a local variable is that a field has a default initial value (p. 82) provided by Java when you do not specify the field's initial value, but a local variable does not.
- The default value for a field of type `String` (or any other reference type) is `null`.
- When a method that specifies a return type (p. 73) is called and completes its task, the method returns a result to its calling method (p. 73).
- Classes often provide `public` methods to allow the class's clients to *set* or *get* `private` instance variables (p. 83). The names of these methods need not begin with *set* or *get*, but this naming convention is recommended and is required for special Java software components called JavaBeans.
- The UML represents instance variables as an attribute name, followed by a colon and the type.
- Private attributes are preceded by a minus sign (–) in the UML.
- The UML indicates an operation's return type by placing a colon and the return type after the parentheses following the operation name.
- UML class diagrams (p. 75) do not specify return types for operations that do not return values.

Section 3.5 Primitive Types vs. Reference Types

- Types in Java are divided into two categories—primitive types and reference types. The primitive types are `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`. All other types are reference types, so classes, which specify the types of objects, are reference types.

- A primitive-type variable can store exactly one value of its declared type at a time.
- Primitive-type instance variables are initialized by default. Variables of types `byte`, `char`, `short`, `int`, `long`, `float` and `double` are initialized to `0`. Variables of type `boolean` are initialized to `false`.
- Reference-type variables (called references; p. 84) store the location of an object in the computer's memory. Such variables refer to objects in the program. The object that's referenced may contain many instance variables and methods.
- Reference-type fields are initialized by default to the value `null`.
- A reference to an object (p. 84) is required to invoke an object's instance methods. A primitive-type variable does not refer to an object and therefore cannot be used to invoke a method.

Section 3.6 Initializing Objects with Constructors

- Keyword `new` requests memory from the system to store an object, then calls the corresponding class's constructor (p. 74) to initialize the object.
- A constructor can be used to initialize an object of a class when the object is created.
- Constructors can specify parameters but cannot specify return types.
- If a class does not define constructors, the compiler provides a default constructor (p. 85) with no parameters, and the class's instance variables are initialized to their default values.
- The UML models constructors in the third compartment of a class diagram. To distinguish a constructor from a class's operations, the UML places the word "constructor" between guillemets (« and »; p. 88) before the constructor's name.

Section 3.7 Floating-Point Numbers and Type `double`

- A floating-point number (p. 88) is a number with a decimal point. Java provides two primitive types for storing floating-point numbers (p. 88) in memory—`float` and `double`. The primary difference between these types is that `double` variables can store numbers with larger magnitude and finer detail (known as the number's precision; p. 88) than `float` variables.
- Variables of type `float` represent single-precision floating-point numbers and have seven significant digits. Variables of type `double` represent double-precision floating-point numbers. These require twice as much memory as `float` variables and provide 15 significant digits—approximately double the precision of `float` variables.
- Floating-point literals (p. 88) are of type `double` by default.
- `Scanner` method `nextDouble` (p. 91) returns a `double` value.
- The format specifier `%f` (p. 90) is used to output values of type `float` or `double`. The format specifier `%.2f` specifies that two digits of precision (p. 90) should be output to the right of the decimal point in the floating-point number.
- The default value for a field of type `double` is `0.0`, and the default value for a field of type `int` is `0`.

Self-Review Exercises

3.1 Fill in the blanks in each of the following:

a) Each class declaration that begins with keyword ________ must be stored in a file that has exactly the same name as the class and ends with the `.java` file-name extension.
b) Keyword ________ in a class declaration is followed immediately by the class's name.
c) Keyword ________ requests memory from the system to store an object, then calls the corresponding class's constructor to initialize the object.
d) Each parameter must specify both a(n) ________ and a(n) ________.
e) By default, classes that are compiled in the same directory are considered to be in the same package, known as the ________.

f) When each object of a class maintains its own copy of an attribute, the field that represents the attribute is also known as a(n) ________.
g) Java provides two primitive types for storing floating-point numbers in memory: ________ and ________.
h) Variables of type `double` represent ________ floating-point numbers.
i) `Scanner` method ________ returns a `double` value.
j) Keyword `public` is an access ________.
k) Return type ________ indicates that a method will not return a value.
l) `Scanner` method ________ reads characters until it encounters a newline character, then returns those characters as a `String`.
m) Class `String` is in package ________.
n) A(n) ________ is not required if you always refer to a class with its fully qualified class name.
o) A(n) ________ is a number with a decimal point, such as 7.33, 0.0975 or 1000.12345.
p) Variables of type `float` represent ________ floating-point numbers.
q) The format specifier ________ is used to output values of type `float` or `double`.
r) Types in Java are divided into two categories—________ types and ________ types.

3.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) By convention, method names begin with an uppercase first letter, and all subsequent words in the name begin with a capital first letter.
b) An `import` declaration is not required when one class in a package uses another in the same package.
c) Empty parentheses following a method name in a method declaration indicate that the method does not require any parameters to perform its task.
d) Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they're declared.
e) A primitive-type variable can be used to invoke a method.
f) Variables declared in the body of a particular method are known as instance variables and can be used in all methods of the class.
g) Every method's body is delimited by left and right braces (`{` and `}`).
h) Primitive-type local variables are initialized by default.
i) Reference-type instance variables are initialized by default to the value `null`.
j) Any class that contains `public static void main( String[] args )` can be used to execute an application.
k) The number of arguments in the method call must match the number of parameters in the method declaration's parameter list.
l) Floating-point values that appear in source code are known as floating-point literals and are type `float` by default.

3.3 What is the difference between a local variable and a field?

3.4 Explain the purpose of a method parameter. What is the difference between a parameter and an argument?

Answers to Self-Review Exercises

3.1 a) `public`. b) `class`. c) `new`. d) type, name. e) default package. f) instance variable. g) `float`, `double`. h) double-precision. i) `nextDouble`. j) modifier. k) `void`. l) `nextLine`. m) `java.lang`. n) `import` declaration. o) floating-point number. p) single-precision. q) `%f`. r) primitive, reference.

3.2 a) False. By convention, method names begin with a lowercase first letter and all subsequent words in the name begin with a capital first letter. b) True. c) True. d) True. e) False. A primitive-type variable cannot be used to invoke a method—a reference to an object is required to invoke the object's methods. f) False. Such variables are called local variables and can be used only in the method in which they're declared. g) True. h) False. Primitive-type instance variables are initialized by default. Each local variable must explicitly be assigned a value. i) True. j) True. k) True. l) False. Such literals are of type `double` by default.

3.3 A local variable is declared in the body of a method and can be used only from the point at which it's declared through the end of the method declaration. A field is declared in a class, but not in the body of any of the class's methods. Also, fields are accessible to all methods of the class. (We'll see an exception to this in Chapter 8, Classes and Objects: A Deeper Look.)

3.4 A parameter represents additional information that a method requires to perform its task. Each parameter required by a method is specified in the method's declaration. An argument is the actual value for a method parameter. When a method is called, the argument values are passed to the corresponding parameters of the method so that it can perform its task.

Exercises

3.5 *(Keyword* **`new`***)* What's the purpose of keyword `new`? Explain what happens when you use it.

3.6 *(Default Constructors)* What is a default constructor? How are an object's instance variables initialized if a class has only a default constructor?

3.7 *(Instance Variables)* Explain the purpose of an instance variable.

3.8 *(Using Classes Without Importing Them)* Most classes need to be imported before they can be used in an application. Why is every application allowed to use classes `System` and `String` without first importing them?

3.9 *(Using a Class Without Importing It)* Explain how a program could use class `Scanner` without importing it.

3.10 *(***set** *and* **get** *Methods)* Explain why a class might provide a *set* method and a *get* method for an instance variable.

3.11 *(Modified* **`GradeBook`** *Class)* Modify class `GradeBook` (Fig. 3.10) as follows:

a) Include a `String` instance variable that represents the name of the course's instructor.
b) Provide a *set* method to change the instructor's name and a *get* method to retrieve it.
c) Modify the constructor to specify two parameters—one for the course name and one for the instructor's name.
d) Modify method `displayMessage` to output the welcome message and course name, followed by `"This course is presented by: "` and the instructor's name.

Use your modified class in a test application that demonstrates the class's new capabilities.

3.12 *(Modified* **`Account`** *Class)* Modify class `Account` (Fig. 3.13) to provide a method called `debit` that withdraws money from an `Account`. Ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the method should print a message indicating `"Debit amount exceeded account balance."` Modify class `AccountTest` (Fig. 3.14) to test method `debit`.

3.13 *(***`Invoice`*** Class)* Create a class called `Invoice` that a hardware store might use to represent an invoice for an item sold at the store. An `Invoice` should include four pieces of information as instance variables—a part number (type `String`), a part description (type `String`), a quantity of the item being purchased (type `int`) and a price per item (`double`). Your class should have a constructor that initializes the four instance variables. Provide a *set* and a *get* method for each instance variable. In addition, provide a method named `getInvoiceAmount` that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as a `double` value. If the quantity is not positive, it should be set to `0`. If the price per item is not positive, it should be set to `0.0`. Write a test application named `InvoiceTest` that demonstrates class `Invoice`'s capabilities.

3.14 *(`Employee` Class)* Create a class called `Employee` that includes three instance variables—a first name (type `String`), a last name (type `String`) and a monthly salary (`double`). Provide a constructor that initializes the three instance variables. Provide a *set* and a *get* method for each instance variable. If the monthly salary is not positive, do not set its value. Write a test application named `EmployeeTest` that demonstrates class `Employee`'s capabilities. Create two `Employee` objects and display each object's *yearly* salary. Then give each `Employee` a 10% raise and display each `Employee`'s yearly salary again.

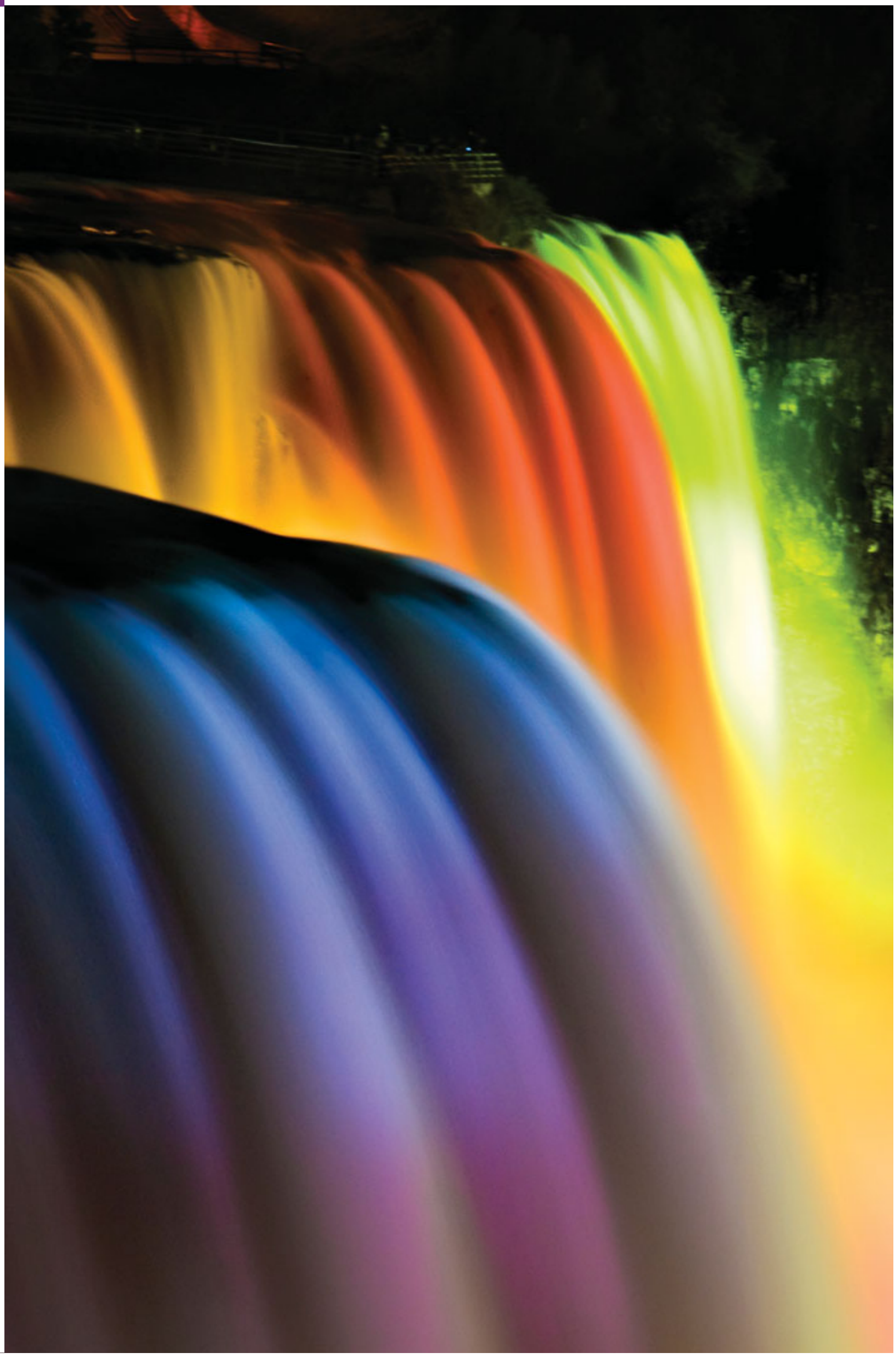
3.15 *(`Date` Class)* Create a class called `Date` that includes three instance variables—a month (type `int`), a day (type `int`) and a year (type `int`). Provide a constructor that initializes the three instance variables and assumes that the values provided are correct. Provide a *set* and a *get* method for each instance variable. Provide a method `displayDate` that displays the month, day and year separated by forward slashes (`/`). Write a test application named `DateTest` that demonstrates class `Date`'s capabilities.

Making a Difference

3.16 *(Target-Heart-Rate Calculator)* While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors. According to the American Heart Association (AHA) (`www.americanheart.org/presenter.jhtml?identifier=4736`), the formula for calculating your *maximum heart rate* in beats per minute is 220 minus your age in years. Your *target heart rate* is a range that's 50–85% of your maximum heart rate. [*Note:* These formulas are estimates provided by the AHA. Maximum and target heart rates may vary based on the health, fitness and gender of the individual. Always consult a physician or qualified health care professional before beginning or modifying an exercise program.] Create a class called `HeartRates`. The class attributes should include the person's first name, last name and date of birth (consisting of separate attributes for the month, day and year of birth). Your class should have a constructor that receives this data as parameters. For each attribute provide *set* and *get* methods. The class also should include a method that calculates and returns the person's age (in years), a method that calculates and returns the person's maximum heart rate and a method that calculates and returns the person's target heart rate. Write a Java application that prompts for the person's information, instantiates an object of class `HeartRates` and prints the information from that object—including the person's first name, last name and date of birth—then calculates and prints the person's age in (years), maximum heart rate and target-heart-rate range.

3.17 *(Computerization of Health Records)* A health care issue that has been in the news lately is the computerization of health records. This possibility is being approached cautiously because of sensitive privacy and security concerns, among others. [We address such concerns in later exercises.] Computerizing health records could make it easier for patients to share their health profiles and histories among their various health care professionals. This could improve the quality of health care, help avoid drug conflicts and erroneous drug prescriptions, reduce costs and, in emergencies, could save lives. In this exercise, you'll design a "starter" `HealthProfile` class for a person. The class attributes should include the person's first name, last name, gender, date of birth (consisting of separate attributes for the month, day and year of birth), height (in inches) and weight (in pounds). Your class should have a constructor that receives this data. For each attribute, provide *set* and *get* methods. The class also should include methods that calculate and return the user's age in years, maximum heart rate and target-heart-rate range (see Exercise 3.16), and body mass index (BMI; see Exercise 2.33). Write a Java application that prompts for the person's information, instantiates an object of class `HealthProfile` for that person and prints the information from that object—including the person's first name, last name, gender, date of birth, height and weight—then calculates and prints the person's age in years, BMI, maximum heart rate and target-heart-rate range. It should also display the BMI values chart from Exercise 2.33.

4 Control Statements: Part 1



Let's all move one place on.
—Lewis Carroll

The wheel is come full circle.
—William Shakespeare

How many apples fell on Newton's head before he took the hint!
—Robert Frost

Objectives

In this chapter you'll learn:

- Basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the `if` and `if...else` selection statements to choose among alternative actions.
- To use the `while` repetition statement to execute statements in a program repeatedly.
- To use counter-controlled repetition and sentinel-controlled repetition.
- To use the compound assignment, increment and decrement operators.
- The portability of primitive data types.

Outline

4.1 Introduction

Before writing a program to solve a problem, you should have a thorough understanding of the problem and a carefully planned approach to solving it. When writing a program, you also should understand the available building blocks and employ proven program-construction techniques. In this chapter and in Chapter 5, Control Statements: Part 2, we discuss these issues in our presentation of the theory and principles of structured programming. The concepts presented here are crucial in building classes and manipulating objects.

We introduce Java's `if`, `if...else` and `while` statements, three of the building blocks that allow you to specify the logic required for methods to perform their tasks. We devote a portion of this chapter (and Chapters 5 and 7) to further developing the `GradeBook` class introduced in Chapter 3. In particular, we add a method to the `GradeBook` class that uses control statements to calculate the average of a set of student grades. Another example demonstrates additional ways to combine control statements to solve a similar problem. We introduce Java's compound assignment, increment and decrement operators. Finally, we discuss the portability of Java's primitive types.

4.2 Algorithms

Any computing problem can be solved by executing a series of actions in a specific order. A procedure for solving a problem in terms of

1. the **actions** to execute and
2. the **order** in which these actions execute

is called an **algorithm**. The following example demonstrates that correctly specifying the order in which the actions execute is important.

Consider the "rise-and-shine algorithm" followed by one executive for getting out of bed and going to work: (1) Get out of bed; (2) take off pajamas; (3) take a shower; (4) get dressed; (5) eat breakfast; (6) carpool to work. This routine gets the executive to work well prepared to make critical decisions. Suppose that the same steps are performed in a slightly different order: (1) Get out of bed; (2) take off pajamas; (3) get dressed; (4) take a shower; (5) eat breakfast; (6) carpool to work. In this case, our executive shows up for work soaking wet. Specifying the order in which statements (actions) execute in a program is called **program control**. This chapter investigates program control using Java's **control statements**.

4.3 Pseudocode

Pseudocode is an informal language that helps you develop algorithms without having to worry about the strict details of Java language syntax. The pseudocode we present is particularly useful for developing algorithms that will be converted to structured portions of Java programs. Pseudocode is similar to everyday English—it's convenient and user friendly, but it's not an actual computer programming language. You'll see an algorithm written in pseudocode in Fig. 4.5.

Pseudocode does not execute on computers. Rather, it helps you "think out" a program before attempting to write it in a programming language, such as Java. This chapter provides several examples of using pseudocode to develop Java programs.

The style of pseudocode we present consists purely of characters, so you can type pseudocode conveniently, using any text-editor program. A carefully prepared pseudocode program can easily be converted to a corresponding Java program.

Pseudocode normally describes only statements representing the actions that occur after you convert a program from pseudocode to Java and the program is run on a computer. Such actions might include input, output or calculations. We typically do not include variable declarations in our pseudocode, but some programmers choose to list variables and mention their purposes at the beginning of their pseudocode.

4.4 Control Structures

Normally, statements in a program are executed one after the other in the order in which they're written. This process is called **sequential execution**. Various Java statements, which we'll soon discuss, enable you to specify that the next statement to execute is *not* necessarily the *next* one in sequence. This is called **transfer of control**.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software development groups. The blame was pointed at the **`goto` statement** (used in most programming languages of the time), which allows you to specify a transfer of control to one of a wide range of destinations in a program. The term **structured programming** became almost synonymous with "`goto` elimination." [*Note:* Java does *not* have a `goto` statement; however, the word `goto` is *reserved* by Java and should *not* be used as an identifier in programs.]

The research of Bohm and Jacopini[1] had demonstrated that programs could be written *without* any `goto` statements. The challenge of the era for programmers was to shift their styles to "`goto`-less programming." Not until the 1970s did most programmers start taking structured programming seriously. The results were impressive. Software development groups reported shorter development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes was that structured programs were clearer, easier to debug and modify, and more likely to be bug free in the first place.

---

1. Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371.

Bohm and Jacopini's work demonstrated that all programs could be written in terms of only three control structures—the **sequence structure**, the **selection structure** and the **repetition structure**. When we introduce Java's control structure implementations, we'll refer to them in the terminology of the *Java Language Specification* as "control statements."

Sequence Structure in Java

The sequence structure is built into Java. Unless directed otherwise, the computer executes Java statements one after the other in the order in which they're written—that is, in sequence. The **activity diagram** in Fig. 4.1 illustrates a typical sequence structure in which two calculations are performed in order. Java lets you have as many actions as you want in a sequence structure. As we'll soon see, anywhere a single action may be placed, we may place several actions in sequence.

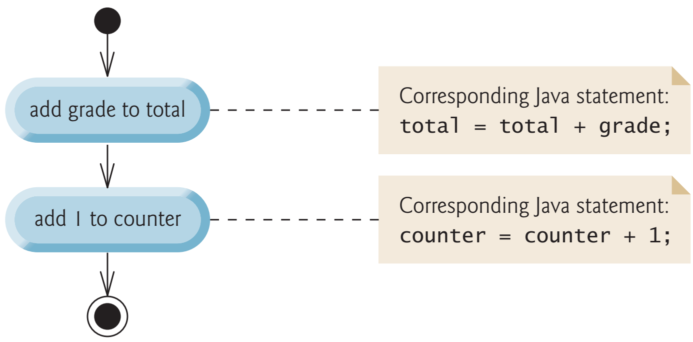


Fig. 4.1 | Sequence structure activity diagram.

A UML activity diagram models the **workflow** (also called the **activity**) of a portion of a software system. Such workflows may include a portion of an algorithm, like the sequence structure in Fig. 4.1. Activity diagrams are composed of symbols, such as **action-state symbols** (rectangles with their left and right sides replaced with outward arcs), **diamonds** and **small circles**. These symbols are connected by **transition arrows**, which represent the flow of the activity—that is, the order in which the actions should occur.

Like pseudocode, activity diagrams help you develop and represent algorithms, although many programmers prefer pseudocode. Activity diagrams clearly show how control structures operate. We use the UML in this chapter and Chapter 5 to show control flow in control statements. In Chapters 12–13, we use the UML in a real-world automated-teller machine case study.

Consider the sequence structure activity diagram in Fig. 4.1. It contains two **action states** that represent actions to perform. Each action state contains an **action expression**—for example, "add grade to total" or "add 1 to counter"—that specifies a particular action to perform. Other actions might include calculations or input/output operations. The arrows in the activity diagram represent **transitions**, which indicate the *order* in which the actions represented by the action states occur. The program that implements the activities illustrated by the diagram in Fig. 4.1 first adds `grade` to `total`, then adds `1` to `counter`.

The **solid circle** at the top of the activity diagram represents the **initial state**—the *beginning* of the workflow *before* the program performs the modeled actions. The **solid**

circle surrounded by a hollow circle that appears at the bottom of the diagram represents the **final state**—the *end* of the workflow *after* the program performs its actions.

Figure 4.1 also includes rectangles with the upper-right corners folded over. These are UML **notes** (like comments in Java)—explanatory remarks that describe the purpose of symbols in the diagram. Figure 4.1 uses UML notes to show the Java code associated with each action state. A **dotted line** connects each note with the element it describes. Activity diagrams normally do *not* show the Java code that implements the activity. We do this here to illustrate how the diagram relates to Java code. For more information on the UML, see our optional case study (Chapters 12–13) or visit `www.uml.org`.

Selection Statements in Java

Java has three types of **selection statements** (discussed in this chapter and Chapter 5). The `if` statement either performs (selects) an action, if a condition is true, or skips it, if the condition is false. The `if…else` statement performs an action if a condition is true and performs a different action if the condition is false. The `switch` statement (Chapter 5) performs one of many different actions, depending on the value of an expression.

The `if` statement is a **single-selection statement** because it selects or ignores a *single* action (or, as we'll soon see, a *single group of actions*). The `if…else` statement is called a **double-selection statement** because it selects between *two different actions* (or *groups of actions*). The `switch` statement is called a **multiple-selection statement** because it selects among *many different actions* (or *groups of actions*).

Repetition Statements in Java

Java provides three **repetition statements** (also called **looping statements**) that enable programs to perform statements repeatedly as long as a condition (called the **loop-continuation condition**) remains true. The repetition statements are the `while`, `do…while` and `for` statements. (Chapter 5 presents the `do…while` and `for` statements.) The `while` and `for` statements perform the action (or group of actions) in their bodies zero or more times—if the loop-continuation condition is initially false, the action (or group of actions) will not execute. The `do…while` statement performs the action (or group of actions) in its body *one or more* times. The words `if`, `else`, `switch`, `while`, `do` and `for` are Java keywords. A complete list of Java keywords appears in Appendix C.

Summary of Control Statements in Java

Java has only three kinds of control structures, which from this point forward we refer to as control statements: the sequence statement, selection statements (three types) and repetition statements (three types). Every program is formed by combining as many of these statements as is appropriate for the algorithm the program implements. We can model each control statement as an activity diagram. Like Fig. 4.1, each diagram contains an initial state and a final state that represent a control statement's entry point and exit point, respectively. **Single-entry/single-exit control statements** make it easy to build programs—we simply connect the exit point of one to the entry point of the next. We call this **control-statement stacking**. We'll learn that there's only one other way in which control statements may be connected—**control-statement nesting**—in which one control statement appears *inside* another. Thus, algorithms in Java programs are constructed from only three kinds of control statements, combined in only two ways. This is the essence of simplicity.

4.5 `if` Single-Selection Statement

Programs use selection statements to choose among alternative courses of action. For example, suppose that the passing grade on an exam is 60. The pseudocode statement

If student's grade is greater than or equal to 60
Print "Passed"

determines whether the condition "student's grade is greater than or equal to 60" is true. If so, "Passed" is printed, and the next pseudocode statement in order is "performed." (Remember, pseudocode is not a real programming language.) If the condition is false, the *Print* statement is ignored, and the next pseudocode statement in order is performed. The indentation of the second line of this selection statement is optional, but recommended, because it emphasizes the inherent structure of structured programs.

The preceding pseudocode *If* statement may be written in Java as

```
if ( studentGrade >= 60 )
   System.out.println( "Passed" );
```

The Java code corresponds closely to the pseudocode. This is one of the properties of pseudocode that makes it such a useful program development tool.

Figure 4.2 illustrates the single-selection `if` statement. This figure contains the most important symbol in an activity diagram—the diamond, or **decision symbol**, which indicates that a decision is to be made. The workflow continues along a path determined by the symbol's associated **guard conditions**, which can be true or false. Each transition arrow emerging from a decision symbol has a guard condition (specified in square brackets next to the arrow). If a guard condition is true, the workflow enters the action state to which the transition arrow points. In Fig. 4.2, if the grade is greater than or equal to 60, the program prints "Passed," then transitions to the activity's final state. If the grade is less than 60, the program immediately transitions to the final state without displaying a message.

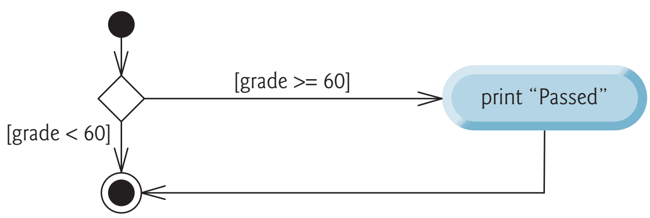


Fig. 4.2 | `if` single-selection statement UML activity diagram.

The `if` statement is a single-entry/single-exit control statement. We'll see that the activity diagrams for the remaining control statements also contain initial states, transition arrows, action states that indicate actions to perform, decision symbols (with associated guard conditions) that indicate decisions to be made, and final states.

4.6 `if...else` Double-Selection Statement

The `if` single-selection statement performs an indicated action only when the condition is `true`; otherwise, the action is skipped. The **`if...else` double-selection statement** allows you to specify an action to perform when the condition is true and a different action when the condition is false. For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
Else
    Print "Failed"
```

prints "Passed" if the student's grade is greater than or equal to 60, but prints "Failed" if it's less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is "performed."

The preceding *If...Else* pseudocode statement can be written in Java as

```java
if ( grade >= 60 )
   System.out.println( "Passed" );
else
   System.out.println( "Failed" );
```

The body of the `else` is also indented. Whatever indentation convention you choose should be applied consistently throughout your programs.

Good Programming Practice 4.1

Indent both body statements of an `if...else` statement. Many IDEs do this for you.

Good Programming Practice 4.2

If there are several levels of indentation, each level should be indented the same additional amount of space.

Figure 4.3 illustrates the flow of control in the `if...else` statement. Once again, the symbols in the UML activity diagram (besides the initial state, transition arrows and final state) represent action states and decisions.

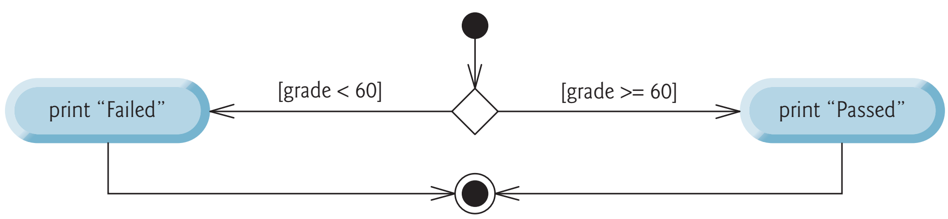


Fig. 4.3 | `if...else` double-selection statement UML activity diagram.

Conditional Operator (?:)

Java provides the **conditional operator** (**`?:`**) that can be used in place of an `if...else` statement. This is Java's only **ternary operator** (operator that takes three operands). Together, the operands and the `?:` symbol form a **conditional expression.** The first operand (to the left of the `?`) is a **`boolean` expression** (i.e., a condition that evaluates to a `boolean` value—**true** or **false**), the second operand (between the ? and :) is the value of the conditional expression if the boolean expression is true and the third operand (to the right of the :) is the value of the conditional expression if the boolean expression evaluates to false. For example, the statement

```
System.out.println( studentGrade >= 60 ? "Passed" : "Failed" );
```

prints the value of println's conditional-expression argument. The conditional expression in this statement evaluates to the string "Passed" if the boolean expression studentGrade >= 60 is true and to the string "Failed" if it's false. Thus, this statement with the conditional operator performs essentially the same function as the if…else statement shown earlier in this section. The precedence of the conditional operator is low, so the entire conditional expression is normally placed in parentheses. We'll see that conditional expressions can be used in some situations where if…else statements cannot.

Nested if…else Statements

A program can test multiple cases by placing if…else statements inside other if…else statements to create **nested if…else statements**. For example, the following pseudocode represents a nested if…else that prints A for exam grades greater than or equal to 90, B for grades 80 to 89, C for grades 70 to 79, D for grades 60 to 69 and F for all other grades:

```
If student's grade is greater than or equal to 90
    Print "A"
else
    If student's grade is greater than or equal to 80
        Print "B"
    else
        If student's grade is greater than or equal to 70
            Print "C"
        else
            If student's grade is greater than or equal to 60
                Print "D"
            else
                Print "F"
```

This pseudocode may be written in Java as

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else
   if ( studentGrade >= 80 )
      System.out.println( "B" );
   else
      if ( studentGrade >= 70 )
         System.out.println( "C" );
      else
         if ( studentGrade >= 60 )
            System.out.println( "D" );
         else
            System.out.println( "F" );
```

If variable `studentGrade` is greater than or equal to 90, the first four conditions in the nested `if…else` statement will be true, but only the statement in the `if` part of the first `if…else` statement will execute. After that statement executes, the `else` part of the "outermost" `if…else` statement is skipped. Many programmers prefer to write the preceding nested `if…else` statement as

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else if ( studentGrade >= 80 )
   System.out.println( "B" );
else if ( studentGrade >= 70 )
   System.out.println( "C" );
else if ( studentGrade >= 60 )
   System.out.println( "D" );
else
   System.out.println( "F" );
```

The two forms are identical except for the spacing and indentation, which the compiler ignores. The latter form avoids deep indentation of the code to the right. Such indentation often leaves little room on a line of source code, forcing lines to be split.

Dangling-`else` Problem

The Java compiler always associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{` and `}`). This behavior can lead to what is referred to as the **dangling-`else` problem**. For example,

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
else
   System.out.println( "x is <= 5" );
```

appears to indicate that if `x` is greater than `5`, the nested `if` statement determines whether `y` is also greater than `5`. If so, the string `"x and y are > 5"` is output. Otherwise, it appears that if `x` is not greater than `5`, the `else` part of the `if…else` outputs the string `"x is <= 5"`. Beware! This nested `if…else` statement does not execute as it appears. The compiler actually interprets the statement as

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
   else
      System.out.println( "x is <= 5" );
```

in which the body of the first `if` is a nested `if…else`. The outer `if` statement tests whether `x` is greater than `5`. If so, execution continues by testing whether `y` is also greater than `5`. If the second condition is true, the proper string—`"x and y are > 5"`—is displayed. However, if the second condition is false, the string `"x is <= 5"` is displayed, even though we know that `x` is greater than `5`. Equally bad, if the outer `if` statement's condition is false, the inner `if…else` is skipped and nothing is displayed.

To force the nested `if…else` statement to execute as it was originally intended, we must write it as follows:

```
if ( x > 5 )
{
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
}
else
   System.out.println( "x is <= 5" );
```

The braces indicate that the second `if` is in the body of the first and that the `else` is associated with the *first* `if`. Exercises 4.27–4.28 investigate the dangling-`else` problem further.

Blocks

The `if` statement normally expects only one statement in its body. To include several statements in the body of an `if` (or the body of an `else` for an `if...else` statement), enclose the statements in braces. Statements contained in a pair of braces form a **block**. A block can be placed anywhere in a program that a single statement can be placed.

The following example includes a block in the `else` part of an `if...else` statement:

```
if ( grade >= 60 )
   System.out.println( "Passed" );
else
{
   System.out.println( "Failed" );
   System.out.println( "You must take this course again." );
}
```

In this case, if `grade` is less than 60, the program executes *both* statements in the body of the `else` and prints

```
Failed
You must take this course again.
```

Note the braces surrounding the two statements in the `else` clause. These braces are important. Without the braces, the statement

```
System.out.println( "You must take this course again." );
```

would be outside the body of the `else` part of the `if...else` statement and would execute *regardless* of whether the grade was less than 60.

Syntax errors (e.g., when one brace in a block is left out of the program) are caught by the compiler. A **logic error** (e.g., when both braces in a block are left out of the program) has its effect at execution time. A **fatal logic error** causes a program to fail and terminate prematurely. A **nonfatal logic error** allows a program to continue executing but causes it to produce incorrect results.

Just as a block can be placed anywhere a single statement can be placed, it's also possible to have an empty statement. Recall from Section 2.8 that the empty statement is represented by placing a semicolon (`;`) where a statement would normally be.

Common Programming Error 4.1

Placing a semicolon after the condition in an `if` or `if...else` statement leads to a logic error in single-selection `if` statements and a syntax error in double-selection `if...else` statements (when the `if`-part contains an actual body statement).

4.7 `while` Repetition Statement

A **repetition** (or **looping**) **statement** allows you to specify that a program should repeat an action while some condition remains true. The pseudocode statement

> *While there are more items on my shopping list*
> *Purchase next item and cross it off my list*

describes the repetition that occurs during a shopping trip. The condition "there are more items on my shopping list" may be true or false. If it's true, then the action "Purchase next item and cross it off my list" is performed. This action will be performed repeatedly while the condition remains true. The statement(s) contained in the *While* repetition statement constitute its body, which may be a single statement or a block. Eventually, the condition will become false (when the last item on the shopping list has been purchased and crossed off). At this point, the repetition terminates, and the first statement after the repetition statement executes.

As an example of Java's **`while` repetition statement**, consider a program segment that finds the first power of 3 larger than 100. Suppose that the `int` variable `product` is initialized to `3`. After the following `while` statement executes, `product` contains the result:

```
while ( product <= 100 )
   product = 3 * product;
```

When this `while` statement begins execution, the value of variable `product` is 3. Each iteration of the `while` statement multiplies `product` by 3, so `product` takes on the values 9, 27, 81 and 243 successively. When variable `product` becomes 243, the `while`-statement condition—`product <= 100`—becomes false. This terminates the repetition, so the final value of `product` is 243. At this point, program execution continues with the next statement after the `while` statement.

Common Programming Error 4.2

Not providing in the body of a `while` statement an action that eventually causes the condition in the `while` to become false normally results in a logic error called an ***infinite loop*** *(the loop never terminates).*

The UML activity diagram in Fig. 4.4 illustrates the flow of control in the preceding `while` statement. Once again, the symbols in the diagram (besides the initial state, transition arrows, a final state and three notes) represent an action state and a decision. This diagram introduces the UML's **merge symbol**. The UML represents both the merge symbol and the decision symbol as diamonds. The merge symbol joins two flows of activity into one. In this diagram, the merge symbol joins the transitions from the initial state and from the action state, so they both flow into the decision that determines whether the loop should begin (or continue) executing. The decision and merge symbols can be distinguished by the number of "incoming" and "outgoing" transition arrows. A decision symbol has one transition arrow pointing to the diamond and two or more pointing out from it to indicate possible transitions from that point. In addition, each transition arrow pointing out of a decision symbol has a guard condition next to it. A merge symbol has two or more transition arrows pointing to the diamond and only one pointing from the diamond, to indicate multiple activity flows merging to continue the activity. *None* of the transition arrows associated with a merge symbol has a guard condition.

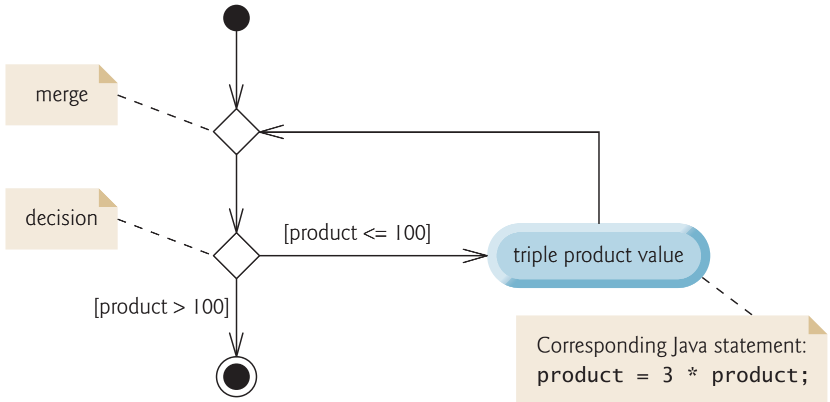


Fig. 4.4 | `while` repetition statement UML activity diagram.

Figure 4.4 clearly shows the repetition of the `while` statement discussed earlier in this section. The transition arrow emerging from the action state points back to the merge, from which program flow transitions back to the decision that's tested at the beginning of each iteration of the loop. The loop continues to execute until the guard condition `product > 100` becomes true. Then the `while` statement exits (reaches its final state), and control passes to the next statement in sequence in the program.

4.8 Formulating Algorithms: Counter-Controlled Repetition

To illustrate how algorithms are developed, we modify the `GradeBook` class of Chapter 3 to solve two variations of a problem that averages student grades. Consider the following problem statement:

> *A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.*

The class average is equal to the sum of the grades divided by the number of students. The algorithm for solving this problem on a computer must input each grade, keep track of the total of all grades input, perform the averaging calculation and print the result.

Pseudocode Algorithm with Counter-Controlled Repetition

Let's use pseudocode to list the actions to execute and specify the order in which they should execute. We use **counter-controlled repetition** to input the grades one at a time. This technique uses a variable called a **counter** (or **control variable**) to control the number of times a set of statements will execute. Counter-controlled repetition is often called **definite repetition**, because the number of repetitions is known *before* the loop begins executing. In this example, repetition terminates when the counter exceeds 10. This section presents a fully developed pseudocode algorithm (Fig. 4.5) and a version of class `GradeBook` (Fig. 4.6) that implements the algorithm in a Java method. We then present an application (Fig. 4.7) that demonstrates the algorithm in action. In Section 4.9, we demonstrate how to use pseudocode to develop such an algorithm from scratch.

Software Engineering Observation 4.1

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, producing a working Java program from the algorithm is usually straightforward.

Note the references in the algorithm of Fig. 4.5 to a total and a counter. A **total** is a variable used to accumulate the sum of several values. A counter is a variable used to count—in this case, the grade counter indicates which of the 10 grades is about to be entered by the user. Variables used to store totals are normally initialized to zero before being used in a program.

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
   Prompt the user to enter the next grade
   Input the next grade
   Add the grade into the total
   Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 4.5 | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

Implementing Counter-Controlled Repetition in Class GradeBook

Class `GradeBook` (Fig. 4.6) contains a constructor (lines 11–14) that assigns a value to the class's instance variable `courseName` (declared in line 8). Lines 17–20, 23–26 and 29–34 declare methods `setCourseName`, `getCourseName` and `displayMessage`, respectively. Lines 37–66 declare method `determineClassAverage`, which implements the class-averaging algorithm described by the pseudocode in Fig. 4.5.

Line 40 declares and initializes `Scanner` variable `input`, which is used to read values entered by the user. Lines 42–45 declare local variables `total`, `gradeCounter`, `grade` and `average` to be of type `int`. Variable `grade` stores the user input.

```
1  // Fig. 4.6: GradeBook.java
2  // GradeBook class that solves class-average problem using
3  // counter-controlled repetition.
4  import java.util.Scanner; // program uses class Scanner
5
6  public class GradeBook
7  {
8     private String courseName; // name of course this GradeBook represents
```

Fig. 4.6 | `GradeBook` class that solves class-average problem using counter-controlled repetition. (Part 1 of 3.)

```java

   // constructor initializes courseName
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

   // determine class average based on 10 grades entered by user
   public void determineClassAverage()
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      int total; // sum of grades entered by user
      int gradeCounter; // number of the grade to be entered next
      int grade; // grade value entered by user
      int average; // average of grades

      // initialization phase
      total = 0; // initialize total
      gradeCounter = 1; // initialize loop counter

      // processing phase uses counter-controlled repetition
      while ( gradeCounter <= 10 ) // loop 10 times
      {
         System.out.print( "Enter grade: " ); // prompt
         grade = input.nextInt(); // input next grade
         total = total + grade; // add grade to total
         gradeCounter = gradeCounter + 1; // increment counter by 1
      } // end while

```

Fig. 4.6 | GradeBook class that solves class-average problem using counter-controlled repetition. (Part 2 of 3.)

```
60         // termination phase
61         average = total / 10; // integer division yields integer result
62
63         // display total and average of grades
64         System.out.printf( "\nTotal of all 10 grades is %d\n", total );
65         System.out.printf( "Class average is %d\n", average );
66      } // end method determineClassAverage
67   } // end class GradeBook
```

Fig. 4.6 | `GradeBook` class that solves class-average problem using counter-controlled repetition. (Part 3 of 3.)

The declarations (in lines 42–45) appear in the body of method `determineClassAverage`. Recall that variables declared in a method body are local variables and can be used only from the line of their declaration to the closing right brace of the method declaration. A local variable's declaration must appear before the variable is used in that method. A local variable cannot be accessed outside the method in which it's declared.

In this chapter, class `GradeBook` simply reads and processes a set of grades. The averaging calculation is performed in method `determineClassAverage` using local variables—we do not preserve any information about student grades in instance variables of the class.

The assignments (in lines 48–49) initialize `total` to `0` and `gradeCounter` to `1`. These initializations occur *before* the variables are used in calculations. Variables `grade` and `average` (for the user input and calculated average, respectively) need not be initialized here—their values will be assigned as they're input or calculated later in the method.

Common Programming Error 4.3

Using the value of a local variable before it's initialized results in a compilation error. All local variables must be initialized before their values are used in expressions.

Error-Prevention Tip 4.1

Initialize each counter and total, either in its declaration or in an assignment statement. Totals are normally initialized to 0. Counters are normally initialized to 0 or 1, depending on how they're used (we'll show examples of when to use 0 and when to use 1).

Line 52 indicates that the `while` statement should continue looping (also called **iterating**) as long as `gradeCounter`'s value is less than or equal to 10. While this condition remains true, the `while` statement repeatedly executes the statements between the braces that delimit its body (lines 54–57).

Line 54 displays the prompt `"Enter grade: "`. Line 55 reads the grade entered by the user and assigns it to variable `grade`. Then line 56 adds the new `grade` entered by the user to the `total` and assigns the result to `total`, which replaces its previous value.

Line 57 adds `1` to `gradeCounter` to indicate that the program has processed a grade and is ready to input the next grade from the user. Incrementing `gradeCounter` eventually causes it to exceed 10. Then the loop terminates, because its condition (line 52) becomes false.

When the loop terminates, line 61 performs the averaging calculation and assigns its result to the variable `average`. Line 64 uses `System.out`'s `printf` method to display the text "`Total of all 10 grades is `" followed by variable `total`'s value. Line 65 then uses `printf` to display the text "`Class average is `" followed by variable `average`'s value. After reaching line 66, method `determineClassAverage` returns control to the calling method (i.e., `main` in `GradeBookTest` of Fig. 4.7).

Class `GradeBookTest`

Class `GradeBookTest` (Fig. 4.7) creates an object of class `GradeBook` (Fig. 4.6) and demonstrates its capabilities. Lines 10–11 of Fig. 4.7 create a new `GradeBook` object and assign it to variable `myGradeBook`. The `String` in line 11 is passed to the `GradeBook` constructor (lines 11–14 of Fig. 4.6). Line 13 calls `myGradeBook`'s `displayMessage` method to display a welcome message to the user. Line 14 then calls `myGradeBook`'s `determineClassAverage` method to allow the user to enter 10 grades, for which the method then calculates and prints the average—the method performs the algorithm shown in Fig. 4.5.

```
 1  // Fig. 4.7: GradeBookTest.java
 2  // Create GradeBook object and invoke its determineClassAverage method.
 3
 4  public class GradeBookTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        // create GradeBook object myGradeBook and
 9        // pass course name to constructor
10        GradeBook myGradeBook = new GradeBook(
11           "CS101 Introduction to Java Programming" );
12
13        myGradeBook.displayMessage(); // display welcome message
14        myGradeBook.determineClassAverage(); // find average of 10 grades
15     } // end main
16  } // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

Fig. 4.7 | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.6) and invokes its `determineClassAverage` method.

Notes on Integer Division and Truncation

The averaging calculation performed by method `determineClassAverage` in response to the method call at line 14 in Fig. 4.7 produces an integer result. The program's output indicates that the sum of the grade values in the sample execution is 846, which, when divided by 10, should yield the floating-point number 84.6. However, the result of the calculation `total / 10` (line 61 of Fig. 4.6) is the integer 84, because `total` and `10` are both integers. Dividing two integers results in **integer division**—any fractional part of the calculation is lost (i.e., **truncated**). In the next section we'll see how to obtain a floating-point result from the averaging calculation.

Common Programming Error 4.4

Assuming that integer division rounds (rather than truncates) can lead to incorrect results. For example, 7 ÷ 4, which yields 1.75 in conventional arithmetic, truncates to 1 in integer arithmetic, rather than rounding to 2.

4.9 Formulating Algorithms: Sentinel-Controlled Repetition

Let's generalize Section 4.8's class-average problem. Consider the following problem:

> *Develop a class-averaging program that processes grades for an arbitrary number of students each time it's run.*

In the previous class-average example, the problem statement specified the number of students, so the number of grades (10) was known in advance. In this example, no indication is given of how many grades the user will enter during the program's execution. The program must process an arbitrary number of grades. How can it determine when to stop the input of grades? How will it know when to calculate and print the class average?

One way to solve this problem is to use a special value called a **sentinel value** (also called a **signal value**, a **dummy value** or a **flag value**) to indicate "end of data entry." The user enters grades until all legitimate grades have been entered. The user then types the sentinel value to indicate that no more grades will be entered. **Sentinel-controlled repetition** is often called **indefinite repetition** because the number of repetitions is *not* known before the loop begins executing.

Clearly, a sentinel value must be chosen that cannot be confused with an acceptable input value. Grades on a quiz are nonnegative integers, so –1 is an acceptable sentinel value for this problem. Thus, a run of the class-average program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The program would then compute and print the class average for the grades 95, 96, 75, 74 and 89; since –1 is the sentinel value, it should *not* enter into the averaging calculation.

Developing the Pseudocode Algorithm with Top-Down, Stepwise Refinement: The Top and First Refinement

We approach this class-average program with a technique called **top-down, stepwise refinement**, which is essential to the development of well-structured programs. We begin with a pseudocode representation of the **top**—a single statement that conveys the overall function of the program:

> *Determine the class average for the quiz*

The top is, in effect, a *complete* representation of a program. Unfortunately, the top rarely conveys sufficient detail from which to write a Java program. So we now begin the refinement process. We divide the top into a series of smaller tasks and list these in the order in which they'll be performed. This results in the following **first refinement**:

Initialize variables
Input, sum and count the quiz grades
Calculate and print the class average

This refinement uses only the sequence structure—the steps listed should execute in order, one after the other.

Software Engineering Observation 4.2

Each refinement, as well as the top itself, is a complete specification of the algorithm—only the level of detail varies.

Software Engineering Observation 4.3

Many programs can be divided logically into three phases: an initialization phase *that initializes the variables; a* processing phase *that inputs data values and adjusts program variables accordingly; and a* termination phase *that calculates and outputs the final results.*

Proceeding to the Second Refinement

The preceding Software Engineering Observation is often all you need for the first refinement in the top-down process. To proceed to the next level of refinement—that is, the **second refinement**—we commit to specific variables. In this example, we need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the value of each grade as it's input by the user and a variable to hold the calculated average. The pseudocode statement

Initialize variables

can be refined as follows:

Initialize total to zero
Initialize counter to zero

Only the variables *total* and *counter* need to be initialized before they're used. The variables *average* and *grade* (for the calculated average and the user input, respectively) need not be initialized, because their values will be replaced as they're calculated or input.

The pseudocode statement

Input, sum and count the quiz grades

requires a repetition structure (i.e., a loop) that successively inputs each grade. We do not know in advance how many grades are to be processed, so we'll use sentinel-controlled repetition. The user enters grades one at a time. After entering the last grade, the user enters the sentinel value. The program tests for the sentinel value after each grade is input and terminates the loop when the user enters the sentinel value. The second refinement of the preceding pseudocode statement is then

```
Prompt the user to enter the first grade
Input the first grade (possibly the sentinel)

While the user has not yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Prompt the user to enter the next grade
    Input the next grade (possibly the sentinel)
```

In pseudocode, we do not use braces around the statements that form the body of the *While* structure. We simply indent the statements under the *While* to show that they belong to the *While*. Again, pseudocode is only an informal program development aid.

The pseudocode statement

```
Calculate and print the class average
```

can be refined as follows:

```
If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print "No grades were entered"
```

We're careful here to test for the possibility of division by zero—a logic error that, if undetected, would cause the program to fail or produce invalid output. The complete second refinement of the pseudocode for the class-average problem is shown in Fig. 4.8.

Error-Prevention Tip 4.2

When performing division by an expression whose value could be zero, test for this and handle it (e.g., print an error message) rather than allow the error to occur.

```
Initialize total to zero
Initialize counter to zero

Prompt the user to enter the first grade
Input the first grade (possibly the sentinel)

While the user has not yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Prompt the user to enter the next grade
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print "No grades were entered"
```

Fig. 4.8 | Class-average problem pseudocode algorithm with sentinel-controlled repetition.

In Fig. 4.5 and Fig. 4.8, we included blank lines and indentation in the pseudocode to make it more readable. The blank lines separate the algorithms into their phases and set off control statements; the indentation emphasizes the bodies of the control statements.

The pseudocode algorithm in Fig. 4.8 solves the more general class-average problem. This algorithm was developed after two refinements. Sometimes more are needed.

Software Engineering Observation 4.4

Terminate the top-down, stepwise refinement process when you've specified the pseudocode algorithm in sufficient detail for you to convert the pseudocode to Java. Normally, implementing the Java program is then straightforward.

Software Engineering Observation 4.5

Some programmers do not use program development tools like pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this may work for simple and familiar problems, it can lead to serious errors and delays in large, complex projects.

Implementing Sentinel-Controlled Repetition in Class GradeBook

Figure 4.9 shows the Java class `GradeBook` containing method `determineClassAverage` that implements the pseudocode algorithm of Fig. 4.8. Although each grade is an integer, the averaging calculation is likely to produce a number with a decimal point—in other words, a real (i.e., floating-point) number. The type `int` cannot represent such a number, so this class uses type `double` to do so.

```java
// Fig. 4.9: GradeBook.java
// GradeBook class that solves the class-average problem using
// sentinel-controlled repetition.
import java.util.Scanner; // program uses class Scanner

public class GradeBook
{
   private String courseName; // name of course this GradeBook represents

   // constructor initializes courseName
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
```

Fig. 4.9 | `GradeBook` class that solves the class-average problem using sentinel-controlled repetition. (Part 1 of 3.)

```java
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

   // determine the average of an arbitrary number of grades
   public void determineClassAverage()
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      int total; // sum of grades
      int gradeCounter; // number of grades entered
      int grade; // grade value
      double average; // number with decimal point for average

      // initialization phase
      total = 0; // initialize total
      gradeCounter = 0; // initialize loop counter

      // processing phase
      // prompt for input and read grade from user
      System.out.print( "Enter grade or -1 to quit: " );
      grade = input.nextInt();

      // loop until sentinel value read from user
      while ( grade != -1 )
      {
         total = total + grade; // add grade to total
         gradeCounter = gradeCounter + 1; // increment counter

         // prompt for input and read next grade from user
         System.out.print( "Enter grade or -1 to quit: " );
         grade = input.nextInt();
      } // end while

      // termination phase
      // if user entered at least one grade...
      if ( gradeCounter != 0 )
      {
         // calculate average of all grades entered
         average = (double) total / gradeCounter;

         // display total and average (with two digits of precision)
         System.out.printf( "\nTotal of the %d grades entered is %d\n",
            gradeCounter, total );
```

Fig. 4.9 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 2 of 3.)

```
77            System.out.printf( "Class average is %.2f\n", average );
78         } // end if
79         else // no grades were entered, so output appropriate message
80            System.out.println( "No grades were entered" );
81      } // end method determineClassAverage
82   } // end class GradeBook
```

Fig. 4.9 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 3 of 3.)

In this example, we see that control statements may be *stacked* on top of one another (in sequence). The `while` statement (lines 57–65) is followed in sequence by an `if...else` statement (lines 69–80). Much of the code in this program is identical to that in Fig. 4.6, so we concentrate on the new concepts.

Line 45 declares `double` variable `average`, which allows us to store the class average as a floating-point number. Line 49 initializes `gradeCounter` to `0`, because no grades have been entered yet. Remember that this program uses sentinel-controlled repetition to input the grades. To keep an accurate record of the number of grades entered, the program increments `gradeCounter` only when the user enters a valid grade.

Program Logic for Sentinel-Controlled Repetition vs. Counter-Controlled Repetition

Compare the program logic for sentinel-controlled repetition in this application with that for counter-controlled repetition in Fig. 4.6. In counter-controlled repetition, each iteration of the `while` statement (e.g., lines 52–58 of Fig. 4.6) reads a value from the user, for the specified number of iterations. In sentinel-controlled repetition, the program reads the first value (lines 53–54 of Fig. 4.9) before reaching the `while`. This value determines whether the program's flow of control should enter the body of the `while`. If the condition of the `while` is false, the user entered the sentinel value, so the body of the `while` does not execute (i.e., no grades were entered). If, on the other hand, the condition is true, the body begins execution, and the loop adds the `grade` value to the `total` (line 59). Then lines 63–64 in the loop body input the next value from the user. Next, program control reaches the closing right brace of the loop body at line 65, so execution continues with the test of the `while`'s condition (line 57). The condition uses the most recent `grade` input by the user to determine whether the loop body should execute again. The value of variable `grade` is always input from the user immediately before the program tests the `while` condition. This allows the program to determine whether the value just input is the sentinel value *before* the program processes that value (i.e., adds it to the `total`). If the sentinel value is input, the loop terminates, and the program does not add –1 to the `total`.

Good Programming Practice 4.3

In a sentinel-controlled loop, prompts should remind the user of the sentinel.

After the loop terminates, the `if...else` statement at lines 69–80 executes. The condition at line 69 determines whether any grades were input. If none were input, the `else` part (lines 79–80) of the `if...else` statement executes and displays the message `"No grades were entered"` and the method returns control to the calling method.

Notice the `while` statement's block in Fig. 4.9 (lines 58–65). Without the braces, the loop would consider its body to be only the first statement, which adds the `grade` to the total. The last three statements in the block would fall outside the loop body, causing the computer to interpret the code incorrectly as follows:

```
while ( grade != -1 )
   total = total + grade; // add grade to total
gradeCounter = gradeCounter + 1; // increment counter

// prompt for input and read next grade from user
System.out.print( "Enter grade or -1 to quit: " );
grade = input.nextInt();
```

The preceding code would cause an infinite loop in the program if the user did not input the sentinel -1 at line 54 (before the while statement).

Common Programming Error 4.5

Omitting the braces that delimit a block can lead to logic errors, such as infinite loops. To prevent this problem, some programmers enclose the body of every control statement in braces, even if the body contains only a single statement.

Explicitly and Implicitly Converting Between Primitive Types

If at least one grade was entered, line 72 of Fig. 4.9 calculates the average of the grades. Recall from Fig. 4.6 that integer division yields an integer result. Even though variable average is declared as a double (line 45), the calculation

```
average = total / gradeCounter;
```

loses the fractional part of the quotient *before* the result of the division is assigned to average. This occurs because total and gradeCounter are *both* integers, and integer division yields an integer result. To perform a floating-point calculation with integer values, we must temporarily treat these values as floating-point numbers for use in the calculation. Java provides the **unary cast operator** to accomplish this task. Line 72 uses the **(double)** cast operator—a unary operator—to create a *temporary* floating-point copy of its operand total (which appears to the right of the operator). Using a cast operator in this manner is called **explicit conversion** or **type casting**. The value stored in total is still an integer.

The calculation now consists of a floating-point value (the temporary double version of total) divided by the integer gradeCounter. Java knows how to evaluate only arithmetic expressions in which the operands' types are *identical*. To ensure that the operands are of the same type, Java performs an operation called **promotion** (or **implicit conversion**) on selected operands. For example, in an expression containing values of the types int and double, the int values are promoted to double values for use in the expression. In this example, the value of gradeCounter is promoted to type double, then the floating-point division is performed and the result of the calculation is assigned to average. As long as the (double) cast operator is applied to *any* variable in the calculation, the calculation will yield a double result. Later in this chapter, we discuss all the primitive types. You'll learn more about the promotion rules in Section 6.7.

Common Programming Error 4.6

A cast operator can be used to convert between primitive numeric types, such as int and double, and between related reference types (as we discuss in Chapter 10, Object-Oriented Programming: Polymorphism). Casting to the wrong type may cause compilation errors or runtime errors.

A cast operator is formed by placing parentheses around any type's name. The operator is a **unary operator** (i.e., an operator that takes only one operand). Java also supports unary versions of the plus (+) and minus (–) operators, so you can write expressions like `-7` or `+5`. Cast operators associate from right to left and have the same precedence as other unary operators, such as unary `+` and unary `-`. This precedence is one level higher than that of the **multiplicative operators** `*`, `/` and `%`. (See the operator precedence chart in Appendix A.) We indicate the cast operator with the notation (*type*) in our precedence charts, to indicate that any type name can be used to form a cast operator.

Line 77 displays the class average. In this example, we display the class average rounded to the nearest hundredth. The format specifier `%.2f` in `printf`'s format control string indicates that variable `average`'s value should be displayed with two digits of precision to the right of the decimal point—indicated by `.2` in the format specifier. The three grades entered during the sample execution of class `GradeBookTest` (Fig. 4.10) total 257, which yields the average 85.666666.... Method `printf` uses the precision in the format specifier to round the value to the specified number of digits. In this program, the average is rounded to the hundredths position and is displayed as `85.67`.

```java
// Fig. 4.10: GradeBookTest.java
// Create GradeBook object and invoke its determineClassAverage method.

public class GradeBookTest
{
   public static void main( String[] args )
   {
      // create GradeBook object myGradeBook and
      // pass course name to constructor
      GradeBook myGradeBook = new GradeBook(
         "CS101 Introduction to Java Programming" );

      myGradeBook.displayMessage(); // display welcome message
      myGradeBook.determineClassAverage(); // find average of grades
   } // end main
} // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

Fig. 4.10 | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.9) and invokes its `determineClassAverage` method.

4.10 Formulating Algorithms: Nested Control Statements

For the next example, we once again formulate an algorithm by using pseudocode and top-down, stepwise refinement, and write a corresponding Java program. We've seen that control statements can be stacked on top of one another (in sequence). In this case study, we examine the only other structured way control statements can be connected—namely, by **nesting** one control statement within another.

Consider the following problem statement:

> *A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.*
>
> *Your program should analyze the results of the exam as follows:*
>
> 1. *Input each test result (i.e., a 1 or a 2). Display the message "Enter result" on the screen each time the program requests another test result.*
> 2. *Count the number of test results of each type.*
> 3. *Display a summary of the test results, indicating the number of students who passed and the number who failed.*
> 4. *If more than eight students passed the exam, print the message "Bonus to instructor!"*

After reading the problem statement carefully, we make the following observations:

1. The program must process test results for 10 students. A counter-controlled loop can be used, because the number of test results is known in advance.
2. Each test result has a numeric value—either a 1 or a 2. Each time it reads a test result, the program must determine whether it's a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it's a 2. (Exercise 4.24 considers the consequences of this assumption.)
3. Two counters are used to keep track of the exam results—one to count the number of students who passed the exam and one to count the number who failed.
4. After the program has processed all the results, it must decide whether more than eight students passed the exam.

Let's proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:

Analyze exam results and decide whether a bonus should be paid

Once again, the top is a *complete* representation of the program, but several refinements are likely to be needed before the pseudocode can evolve naturally into a Java program.

Our first refinement is

Initialize variables
Input the 10 exam results, and count passes and failures
Print a summary of the exam results and decide whether a bonus should be paid

Here, too, even though we have a *complete* representation of the entire program, further refinement is necessary. We now commit to specific variables. Counters are needed to record the passes and failures, a counter will be used to control the looping process and a variable is needed to store the user input. The variable in which the user input will be stored is *not* initialized at the start of the algorithm, because its value is read from the user during each iteration of the loop.

The pseudocode statement

Initialize variables

can be refined as follows:

Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

Notice that only the counters are initialized at the start of the algorithm.

The pseudocode statement

Input the 10 exam results, and count passes and failures

requires a loop that successively inputs the result of each exam. We know in advance that there are precisely 10 exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., *nested* within the loop), a double-selection structure will determine whether each exam result is a pass or a failure and will increment the appropriate counter. The refinement of the preceding pseudocode statement is then

```
While student counter is less than or equal to 10
    Prompt the user to enter the next exam result
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

    Add one to student counter
```

We use blank lines to isolate the *If…Else* control structure, which improves readability.

The pseudocode statement

Print a summary of the exam results and decide whether a bonus should be paid

can be refined as follows:

```
Print the number of passes
Print the number of failures

If more than eight students passed
    Print "Bonus to instructor!"
```

Complete Second Refinement of Pseudocode and Conversion to Class **`Analysis`**

The complete second refinement appears in Fig. 4.11. Notice that blank lines are also used to set off the *While* structure for program readability. This pseudocode is now sufficiently refined for conversion to Java.

The Java class that implements the pseudocode algorithm and two sample executions are shown in Fig. 4.12. Lines 13–16 of `main` declare the variables that method `processExamResults` of class `Analysis` uses to process the examination results. Several of these

```
Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

While student counter is less than or equal to 10
    Prompt the user to enter the next exam result
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

    Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
    Print "Bonus to instructor!"
```

Fig. 4.11 | Pseudocode for examination-results problem.

declarations use Java's ability to incorporate variable initialization into declarations (`passes` is assigned `0`, `failures 0` and `studentCounter 1`). Looping programs may require initialization at the beginning of each repetition—normally performed by assignment statements rather than in declarations.

Error-Prevention Tip 4.3

Initializing local variables when they're declared helps you avoid any compilation errors that might arise from attempts to use uninitialized variables. While Java does not require that local-variable initializations be incorporated into declarations, it does require that local variables be initialized before their values are used in an expression.

The `while` statement (lines 19–33) loops 10 times. During each iteration, the loop inputs and processes one exam result. Notice that the `if...else` statement (lines 26–29) for processing each result is nested in the `while` statement. If the `result` is `1`, the `if...else` statement increments `passes`; otherwise, it assumes the `result` is `2` and increments `failures`. Line 32 increments `studentCounter` before the loop condition is tested again at line 19. After 10 values have been input, the loop terminates and line 36 displays the number of `passes` and `failures`. The `if` statement at lines 39–40 determines whether more than eight students passed the exam and, if so, outputs the message `"Bonus to instructor!"`.

```
1 // Fig. 4.12: Analysis.java
2 // Analysis of examination results using nested control statements.
3 import java.util.Scanner; // class uses class Scanner
```

Fig. 4.12 | Analysis of examination results using nested control statements. (Part 1 of 3.)

```java

public class Analysis
{
   public static void main( String[] args )
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      // initializing variables in declarations
      int passes = 0; // number of passes
      int failures = 0; // number of failures
      int studentCounter = 1; // student counter
      int result; // one exam result (obtains value from user)

      // process 10 students using counter-controlled loop
      while ( studentCounter <= 10 )
      {
         // prompt user for input and obtain value from user
         System.out.print( "Enter result (1 = pass, 2 = fail): " );
         result = input.nextInt();

         // if...else is nested in the while statement
         if ( result == 1 )          // if result 1,
            passes = passes + 1;     // increment passes;
         else                        // else result is not 1, so
            failures = failures + 1; // increment failures

         // increment studentCounter so loop eventually terminates
         studentCounter = studentCounter + 1;
      } // end while

      // termination phase; prepare and display results
      System.out.printf( "Passed: %d\nFailed: %d\n", passes, failures );

      // determine whether more than 8 students passed
      if ( passes > 8 )
         System.out.println( "Bonus to instructor!" );
   } // end main
} // end class Analysis
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

Fig. 4.12 | Analysis of examination results using nested control statements. (Part 2 of 3.)

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

Fig. 4.12 | Analysis of examination results using nested control statements. (Part 3 of 3.)

Figure 4.12 shows the input and output from two sample excutions of the program. During the first, the condition at line 39 of method `main` is `true`—more than eight students passed the exam, so the program outputs a message to bonus the instructor.

This example contains only one class, with method `main` performing all the class's work. In this chapter and in Chapter 3, you've seen examples consisting of two classes—one containing methods that perform useful tasks and one containing method `main`, which creates an object of the other class and calls its methods. Occasionally, when it does not make sense to try to create a reusable class to demonstrate a concept, we'll place the program's statements entirely within the `main` method of a single class.

4.11 Compound Assignment Operators

The **compound assignment operators** abbreviate assignment expressions. Statements like

variable = *variable operator expression*;

where *operator* is one of the binary operators `+`, `-`, `*`, `/` or `%` (or others we discuss later in the text) can be written in the form

variable operator= *expression*;

For example, you can abbreviate the statement

```
c = c + 3;
```

with the **addition compound assignment operator, `+=`**, as

```
c += 3;
```

The `+=` operator adds the value of the expression on its right to the value of the variable on its left and stores the result in the variable on the left of the operator. Thus, the assignment expression `c += 3` adds `3` to `c`. Figure 4.13 shows the arithmetic compound assignment operators, sample expressions using the operators and explanations of what the operators do.

4.12 Increment and Decrement Operators

Java provides two unary operators (summarized in Fig. 4.14) for adding 1 to or subtracting 1 from the value of a numeric variable. These are the unary **increment operator, `++`**, and the unary **decrement operator, `--`**. A program can increment by 1 the value of a variable

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to `c`
`-=`	`d -= 4`	`d = d - 4`	1 to `d`
`*=`	`e *= 5`	`e = e * 5`	20 to `e`
`/=`	`f /= 3`	`f = f / 3`	2 to `f`
`%=`	`g %= 9`	`g = g % 9`	3 to `g`

Fig. 4.13 | Arithmetic compound assignment operators.

called `c` using the increment operator, `++`, rather than the expression `c = c + 1` or `c += 1`. An increment or decrement operator that's prefixed to (placed before) a variable is referred to as the **prefix increment** or **prefix decrement operator**, respectively. An increment or decrement operator that's postfixed to (placed after) a variable is referred to as the **postfix increment** or **postfix decrement operator**, respectively.

Operator	Operator name	Sample expression	Explanation
`++`	prefix increment	`++a`	Increment `a` by 1, then use the new value of `a` in the expression in which `a` resides.
`++`	postfix increment	`a++`	Use the current value of `a` in the expression in which `a` resides, then increment `a` by 1.
`--`	prefix decrement	`--b`	Decrement `b` by 1, then use the new value of `b` in the expression in which `b` resides.
`--`	postfix decrement	`b--`	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by 1.

Fig. 4.14 | Increment and decrement operators.

Using the prefix increment (or decrement) operator to add 1 to (or subtract 1 from) a variable is known as **preincrementing** (or **predecrementing**). This causes the variable to be incremented (decremented) by 1; then the new value of the variable is used in the expression in which it appears. Using the postfix increment (or decrement) operator to add 1 to (or subtract 1 from) a variable is known as **postincrementing** (or **postdecrementing**). This causes the current value of the variable to be used in the expression in which it appears; then the variable's value is incremented (decremented) by 1.

Good Programming Practice 4.4

Unlike binary operators, the unary increment and decrement operators should be placed next to their operands, with no intervening spaces.

Figure 4.15 demonstrates the difference between the prefix increment and postfix increment versions of the `++` increment operator. The decrement operator (`--`) works similarly.

```
1  // Fig. 4.15: Increment.java
2  // Prefix increment and postfix increment operators.
3
4  public class Increment
5  {
6     public static void main( String[] args )
7     {
8        int c;
9
10       // demonstrate postfix increment operator
11       c = 5; // assign 5 to c
12       System.out.println( c );   // prints 5
13       System.out.println( c++ ); // prints 5 then postincrements
14       System.out.println( c );   // prints 6
15
16       System.out.println(); // skip a line
17
18       // demonstrate prefix increment operator
19       c = 5; // assign 5 to c
20       System.out.println( c );   // prints 5
21       System.out.println( ++c ); // preincrements then prints 6
22       System.out.println( c );   // prints 6
23    } // end main
24 } // end class Increment
```

```
5
5
6

5
6
6
```

Fig. 4.15 | Preincrementing and postincrementing.

Line 11 initializes the variable c to 5, and line 12 outputs c's initial value. Line 13 outputs the value of the expression c++. This expression postincrements the variable c, so c's original value (5) is output, then c's value is incremented (to 6). Thus, line 13 outputs c's initial value (5) again. Line 14 outputs c's new value (6) to prove that the variable's value was indeed incremented in line 13.

Line 19 resets c's value to 5, and line 20 outputs c's value. Line 21 outputs the value of the expression ++c. This expression preincrements c, so its value is incremented; then the new value (6) is output. Line 22 outputs c's value again to show that the value of c is still 6 after line 21 executes.

The arithmetic compound assignment operators and the increment and decrement operators can be used to simplify program statements. For example, the three assignment statements in Fig. 4.12 (lines 27, 29 and 32)

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

can be written more concisely with compound assignment operators as

```
passes += 1;
failures += 1;
studentCounter += 1;
```

with prefix increment operators as

```
++passes;
++failures;
++studentCounter;
```

or with postfix increment operators as

```
passes++;
failures++;
studentCounter++;
```

When incrementing or decrementing a variable in a statement by itself, the prefix increment and postfix increment forms have the same effect, and the prefix decrement and postfix decrement forms have the same effect. It's only when a variable appears in the context of a larger expression that preincrementing and postincrementing the variable have different effects (and similarly for predecrementing and postdecrementing).

Common Programming Error 4.7

Attempting to use the increment or decrement operator on an expression other than one to which a value can be assigned is a syntax error. For example, writing `++(x + 1)` *is a syntax error, because* `(x + 1)` *is not a variable.*

Figure 4.16 shows the precedence and associativity of the operators we've introduced. They're shown from top to bottom in decreasing order of precedence. The second column describes the associativity of the operators at each level of precedence. The conditional operator (`?:`); the unary operators increment (`++`), decrement (`--`), plus (`+`) and minus (`-`); the cast operators and the assignment operators `=`, `+=`, `-=`, `*=`, `/=` and `%=` associate from right to left. All the other operators in the operator precedence chart in Fig. 4.16 associate from left to right. The third column lists the type of each group of operators.

Operators	Associativity	Type
`++` `--`	right to left	unary postfix
`++` `--` `+` `-` `(` *type* `)`	right to left	unary prefix
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment

Fig. 4.16 | Precedence and associativity of the operators discussed so far.

4.13 Primitive Types

The table in Appendix D lists the eight primitive types in Java. Like its predecessor languages C and C++, Java requires all variables to have a type. For this reason, Java is referred to as a **strongly typed language**.

In C and C++, programmers frequently have to write separate versions of programs to support different computer platforms, because the primitive types are not guaranteed to be identical from computer to computer. For example, an `int` value on one machine might be represented by 16 bits (2 bytes) of memory, on a second machine by 32 bits (4 bytes) of memory, and on another machine by 64 bits (8 bytes) of memory. In Java, `int` values are always 32 bits (4 bytes).

Portability Tip 4.1

The primitive types in Java are portable across all computer platforms that support Java.

Each type in Appendix D is listed with its size in bits (there are eight bits to a byte) and its range of values. Because the designers of Java want to ensure portability, they use internationally recognized standards for character formats (Unicode; for more information, visit `www.unicode.org`) and floating-point numbers (IEEE 754; for more information, visit `grouper.ieee.org/groups/754/`).

Recall from Section 3.4 that variables of primitive types declared outside of a method as fields of a class are automatically assigned default values unless explicitly initialized. Instance variables of types `char`, `byte`, `short`, `int`, `long`, `float` and `double` are all given the value `0` by default. Instance variables of type `boolean` are given the value `false` by default. Reference-type instance variables are initialized by default to the value `null`.

4.14 (Optional) GUI and Graphics Case Study: Creating Simple Drawings

An appealing feature of Java is its graphics support, which enables you to visually enhance your applications. We now introduce one of Java's graphical capabilities—drawing lines. It also covers the basics of creating a window to display a drawing on the computer screen.

Java's Coordinate System

To draw in Java, you must understand Java's **coordinate system** (Fig. 4.17), a scheme for identifying points on the screen. By default, the upper-left corner of a GUI component

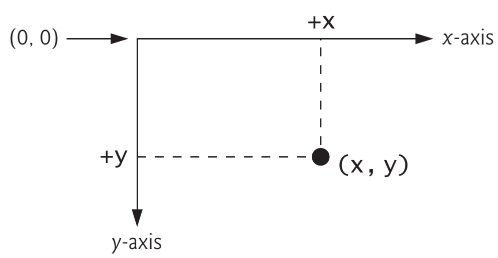


Fig. 4.17 | Java coordinate system. Units are measured in pixels.

has the coordinates (0, 0). A coordinate pair is composed of an **x-coordinate** (the **horizontal coordinate**) and a **y-coordinate** (the **vertical coordinate**). The *x*-coordinate is the horizontal location moving from left to right. The *y*-coordinate is the vertical location moving from top to bottom. The **x-axis** describes every horizontal coordinate, and the **y-axis** every vertical coordinate.

Coordinates indicate where graphics should be displayed on a screen. Coordinate units are measured in **pixels**. The term pixel stands for “picture element.” A pixel is a display monitor’s smallest unit of resolution.

First Drawing Application

Our first drawing application simply draws two lines. Class `DrawPanel` (Fig. 4.18) performs the actual drawing, while class `DrawPanelTest` (Fig. 4.19) creates a window to display the drawing. In class `DrawPanel`, the `import` statements in lines 3–4 allow us to use class **`Graphics`** (from package `java.awt`), which provides various methods for drawing text and shapes onto the screen, and class **`JPanel`** (from package `javax.swing`), which provides an area on which we can draw.

```
1  // Fig. 4.18: DrawPanel.java
2  // Using drawLine to connect the corners of a panel.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5
6  public class DrawPanel extends JPanel
7  {
8     // draws an X from the corners of the panel
9     public void paintComponent( Graphics g )
10    {
11       // call paintComponent to ensure the panel displays correctly
12       super.paintComponent( g );
13
14       int width = getWidth(); // total width
15       int height = getHeight(); // total height
16
17       // draw a line from the upper-left to the lower-right
18       g.drawLine( 0, 0, width, height );
19
20       // draw a line from the lower-left to the upper-right
21       g.drawLine( 0, height, width, 0 );
22    } // end method paintComponent
23 } // end class DrawPanel
```

Fig. 4.18 | Using `drawLine` to connect the corners of a panel.

```
// Fig. 4.19: DrawPanelTest.java
// Application to display a DrawPanel.
import javax.swing.JFrame;

public class DrawPanelTest
{
```

Fig. 4.19 | Creating `JFrame` to display `DrawPanel`. (Part 1 of 2.)

```java
    public static void main( String[] args )
    {
       // create a panel that contains our drawing
       DrawPanel panel = new DrawPanel();

       // create a new frame to hold the panel
       JFrame application = new JFrame();

       // set the frame to exit when it is closed
       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

       application.add( panel ); // add the panel to the frame
       application.setSize( 250, 250 ); // set the size of the frame
       application.setVisible( true ); // make the frame visible
    } // end main
 } // end class DrawPanelTest
```

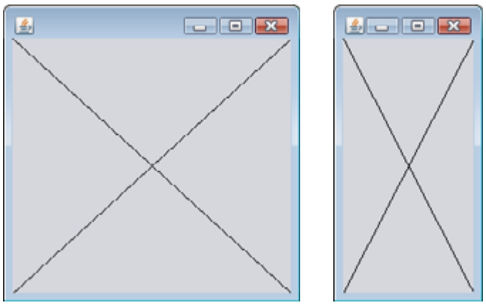


Fig. 4.19 | Creating `JFrame` to display `DrawPanel`. (Part 2 of 2.)

Line 6 uses the keyword **`extends`** to indicate that class `DrawPanel` is an enhanced type of `JPanel`. The keyword `extends` represents a so-called inheritance relationship in which our new class `DrawPanel` begins with the existing members (data and methods) from class `JPanel`. The class from which `DrawPanel` **inherits**, `JPanel`, appears to the right of keyword `extends`. In this inheritance relationship, `JPanel` is called the **superclass** and `DrawPanel` is called the **subclass**. This results in a `DrawPanel` class that has the attributes (data) and behaviors (methods) of class `JPanel` as well as the new features we're adding in our `DrawPanel` class declaration—specifically, the ability to draw two lines along the diagonals of the panel. Inheritance is explained in detail in Chapter 9. For now, you should mimic our `DrawPanel` class when creating your own graphics programs.

Method `paintComponent`

Every `JPanel`, including our `DrawPanel`, has a **`paintComponent`** method (lines 9–22), which the system automatically calls every time it needs to display the `JPanel`. Method `paintComponent` must be declared as shown in line 9—otherwise, the system will not call it. This method is called when a `JPanel` is first displayed on the screen, when it's covered then uncovered by a window on the screen and when the window in which it appears is resized. Method `paintComponent` requires one argument, a `Graphics` object, that's provided by the system when it calls `paintComponent`.

The first statement in every `paintComponent` method you create should always be

```java
super.paintComponent( g );
```

which ensures that the panel is properly rendered before we begin drawing on it. Next, lines 14–15 call methods that class `DrawPanel` inherits from `JPanel`. Because `DrawPanel` extends `JPanel`, `DrawPanel` can use any `public` methods of `JPanel`. Methods **`getWidth`** and **`getHeight`** return the `JPanel`'s width and height, respectively. Lines 14–15 store these values in the local variables `width` and `height`. Finally, lines 18 and 21 use the `Graphics` variable `g` to call method **`drawLine`** to draw the two lines. Method `drawLine` draws a line between two points represented by its four arguments. The first two arguments are the *x*- and *y*-coordinates for one endpoint, and the last two arguments are the coordinates for the other endpoint. If you resize the window, the lines will scale accordingly, because the arguments are based on the width and height of the panel. Resizing the window in this application causes the system to call `paintComponent` to redraw the `DrawPanel`'s contents.

Class `DrawPanelTest`

To display the `DrawPanel` on the screen, you must place it in a window. You create a window with an object of class **`JFrame`**. In `DrawPanelTest.java` (Fig. 4.19), line 3 imports class `JFrame` from package `javax.swing`. Line 10 in `main` creates a `DrawPanel` object, which contains our drawing, and line 13 creates a new `JFrame` that can hold and display our panel. Line 16 calls `JFrame` method **`setDefaultCloseOperation`** with the argument **`JFrame.EXIT_ON_CLOSE`** to indicate that the application should terminate when the user closes the window. Line 18 uses class `JFrame`'s **`add` method** to attach the `DrawPanel` to the `JFrame`. Line 19 sets the size of the `JFrame`. Method **`setSize`** takes two parameters that represent the width and height of the `JFrame`, respectively. Finally, line 20 displays the `JFrame` by calling its **`setVisible` method** with the argument `true`. When the `JFrame` is displayed, the `DrawPanel`'s `paintComponent` method (lines 9–22 of Fig. 4.18) is implicitly called, and the two lines are drawn (see the sample outputs in Fig. 4.19). Try resizing the window to see that the lines always draw based on the window's current width and height.

GUI and Graphics Case Study Exercises

4.1 Using loops and control statements to draw lines can lead to many interesting designs.

a) Create the design in the left screen capture of Fig. 4.20. This design draws lines from the top-left corner, fanning them out until they cover the upper-left half of the panel. One approach is to divide the width and height into an equal number of steps (we found 15 steps worked well). The first endpoint of a line will always be in the top-left corner (0, 0). The second endpoint can be found by starting at the bottom-left corner and moving up one vertical step and right one horizontal step. Draw a line between the two endpoints. Continue moving up and to the right one step to find each successive endpoint. The figure should scale accordingly as you resize the window.

b) Modify part (a) to have lines fan out from all four corners, as shown in the right screen capture of Fig. 4.20. Lines from opposite corners should intersect along the middle.

4.2 Figure 4.21 displays two additional designs created using `while` loops and `drawLine`.

a) Create the design in the left screen capture of Fig. 4.21. Begin by dividing each edge into an equal number of increments (we chose 15 again). The first line starts in the top-left corner and ends one step right on the bottom edge. For each successive line, move down one increment on the left edge and right one increment on the bottom edge. Continue drawing lines until you reach the bottom-right corner. The figure should scale as you resize the window so that the endpoints always touch the edges.

b) Modify your answer in part (a) to mirror the design in all four corners, as shown in the right screen capture of Fig. 4.21.

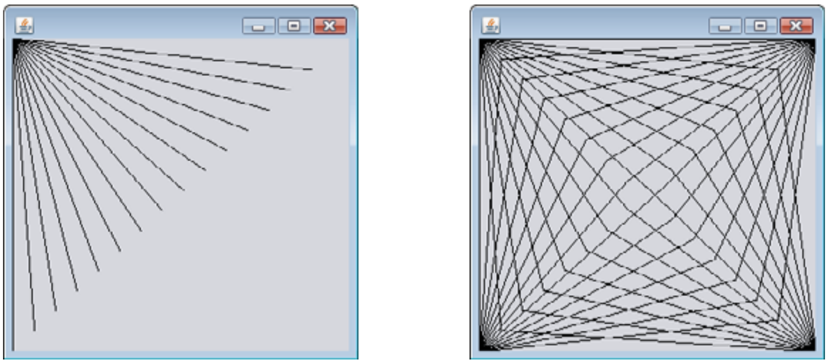


Fig. 4.20 | Lines fanning from a corner.

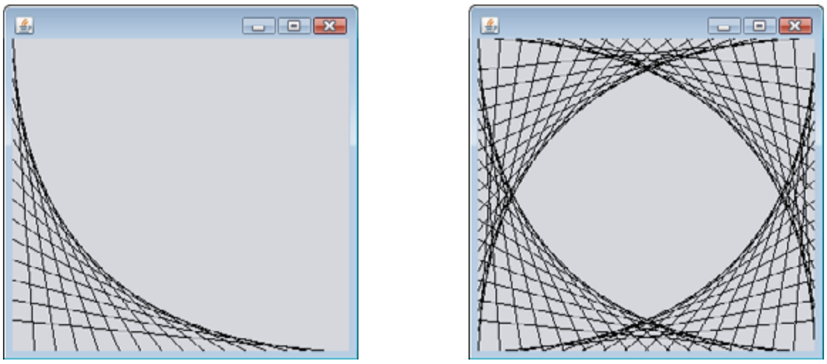


Fig. 4.21 | Line art with loops and `drawLine`.

4.15 Wrap-Up

This chapter presented basic problem solving for building classes and developing methods for these classes. We demonstrated how to construct an algorithm (i.e., an approach to solving a problem), then how to refine the algorithm through several phases of pseudocode development, resulting in Java code that can be executed as part of a method. The chapter showed how to use top-down, stepwise refinement to plan out the specific actions that a method must perform and the order in which the method must perform these actions.

Only three types of control structures—sequence, selection and repetition—are needed to develop any problem-solving algorithm. Specifically, this chapter demonstrated the `if` single-selection statement, the `if...else` double-selection statement and the `while` repetition statement. These are some of the building blocks used to construct solutions to many problems. We used control-statement stacking to total and compute the average of a set of student grades with counter- and sentinel-controlled repetition, and we used control-statement nesting to analyze and make decisions based on a set of exam results. We introduced Java's compound assignment operators and its increment and decrement operators. Finally, we discussed Java's primitive types. In Chapter 5, we continue our discussion of control statements, introducing the `for`, `do...while` and `switch` statements.

Summary

Section 4.1 Introduction

- Before writing a program to solve a problem, you must have a thorough understanding of the problem and a carefully planned approach to solving it. You must also understand the building blocks that are available and employ proven program-construction techniques.

Section 4.2 Algorithms

- Any computing problem can be solved by executing a series of actions (p. 103) in a specific order.
- A procedure for solving a problem in terms of the actions to execute and the order in which they execute is called an algorithm (p. 103).
- Specifying the order in which statements execute in a program is called program control (p. 104).

Section 4.3 Pseudocode

- Pseudocode (p. 104) is an informal language that helps you develop algorithms without having to worry about the strict details of Java language syntax.
- Pseudocode is similar to everyday English—it's convenient and user friendly, but it's not an actual computer programming language.
- Pseudocode helps you "think out" a program before attempting to write it in a programming language, such as Java.
- Carefully prepared pseudocode can easily be converted to a corresponding Java program.

Section 4.4 Control Structures

- Normally, statements in a program are executed one after the other in the order in which they're written. This process is called sequential execution (p. 104).
- Various Java statements enable you to specify that the next statement to execute is not necessarily the next one in sequence. This is called transfer of control (p. 104).
- Bohm and Jacopini demonstrated that all programs could be written in terms of only three control structures (p. 105)—the sequence structure, the selection structure and the repetition structure.
- The term "control structures" comes from the field of computer science. The *Java Language Specification* refers to "control structures" as "control statements" (p. 104).
- The sequence structure is built into Java. Unless directed otherwise, the computer executes Java statements one after the other in the order in which they're written—that is, in sequence.
- Anywhere a single action may be placed, several actions may be placed in sequence.
- Activity diagrams (p. 105) are part of the UML. An activity diagram models the workflow (p. 105; also called the activity) of a portion of a software system.
- Activity diagrams are composed of symbols (p. 105)—such as action-state symbols, diamonds and small circles—that are connected by transition arrows, which represent the flow of the activity.
- Action states (p. 105) contain action expressions that specify particular actions to perform.
- The arrows in an activity diagram represent transitions, which indicate the order in which the actions represented by the action states occur.
- The solid circle located at the top of an activity diagram represents the activity's initial state (p. 105)—the beginning of the workflow before the program performs the modeled actions.
- The solid circle surrounded by a hollow circle that appears at the bottom of the diagram represents the final state (p. 106)—the end of the workflow after the program performs its actions.
- Rectangles with their upper-right corners folded over are UML notes (p. 106)—explanatory remarks that describe the purpose of symbols in the diagram.

- Java has three types of selection statements (p. 106).
- The `if` single-selection statement (p. 106) selects or ignores one or more actions.
- The `if…else` double-selection statement selects between two actions or groups of actions.
- The `switch` statement is called a multiple-selection statement (p. 106) because it selects among many different actions or groups of actions.
- Java provides the `while`, `do…while` and `for` repetition (looping) statements that enable programs to perform statements repeatedly as long as a loop-continuation condition remains true.
- The `while` and `for` statements perform the action(s) in their bodies zero or more times—if the loop-continuation condition (p. 106) is initially false, the action(s) will not execute. The `do…while` statement performs the action(s) in its body one or more times.
- The words `if`, `else`, `switch`, `while`, `do` and `for` are Java keywords. Keywords cannot be used as identifiers, such as variable names.
- Every program is formed by combining as many sequence, selection and repetition statements (p. 106) as is appropriate for the algorithm the program implements.
- Single-entry/single-exit control statements (p. 106) are attached to one another by connecting the exit point of one to the entry point of the next. This is known as control-statement stacking.
- A control statement may also be nested (p. 106) inside another control statement.

Section 4.5 `if` Single-Selection Statement

- Programs use selection statements to choose among alternative courses of action.
- The single-selection `if` statement's activity diagram contains the diamond symbol, which indicates that a decision is to be made. The workflow follows a path determined by the symbol's associated guard conditions (p. 107). If a guard condition is true, the workflow enters the action state to which the corresponding transition arrow points.
- The `if` statement is a single-entry/single-exit control statement.

Section 4.6 `if…else` Double-Selection Statement

- The `if` single-selection statement performs an indicated action only when the condition is `true`.
- The `if…else` double-selection (p. 106) statement performs one action when the condition is true and a different action when the condition is false.
- The conditional operator (p. 108; `?:`) is Java's only ternary operator—it takes three operands. Together, the operands and the `?:` symbol form a conditional expression (p. 108).
- A program can test multiple cases with nested `if…else` statements (p. 109).
- The Java compiler associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces.
- The `if` statement expects one statement in its body. To include several statements in the body of an `if` (or the body of an `else` for an `if…else` statement), enclose the statements in braces.
- A block (p. 111) of statements can be placed anywhere that a single statement can be placed.
- A logic error (p. 111) has its effect at execution time. A fatal logic error (p. 111) causes a program to fail and terminate prematurely. A nonfatal logic error (p. 111) allows a program to continue executing, but causes the program to produce incorrect results.
- Just as a block can be placed anywhere a single statement can be placed, you can also use an empty statement, represented by placing a semicolon (`;`) where a statement would normally be.

Section 4.7 `while` Repetition Statement

- The `while` repetition statement (p. 112) allows you to specify that a program should repeat an action while some condition remains true.

- The UML's merge (p. 112) symbol joins two flows of activity into one.
- The decision and merge symbols can be distinguished by the number of "incoming" and "outgoing" transition arrows. A decision symbol has (p. 107) one transition arrow pointing to the diamond and two or more transition arrows pointing out from the diamond to indicate possible transitions from that point. Each transition arrow pointing out of a decision symbol has a guard condition. A merge symbol has two or more transition arrows pointing to the diamond and only one transition arrow pointing from the diamond, to indicate multiple activity flows merging to continue the activity. None of the transition arrows associated with a merge symbol has a guard condition.

Section 4.8 Formulating Algorithms: Counter-Controlled Repetition

- Counter-controlled repetition (p. 113) uses a variable called a counter (or control variable) to control the number of times a set of statements execute.
- Counter-controlled repetition is often called definite repetition (p. 113), because the number of repetitions is known before the loop begins executing.
- A total (p. 114) is a variable used to accumulate the sum of several values. Variables used to store totals are normally initialized to zero before being used in a program.
- A local variable's declaration must appear before the variable is used in that method. A local variable cannot be accessed outside the method in which it's declared.
- Dividing two integers results in integer division (p. 118)—the calculation's fractional part is truncated.

Section 4.9 Formulating Algorithms: Sentinel-Controlled Repetition

- In sentinel-controlled repetition (p. 118), a special value called a sentinel value (also called a signal value, a dummy value or a flag value) is used to indicate "end of data entry."
- A sentinel value must be chosen that cannot be confused with an acceptable input value.
- Top-down, stepwise refinement (p. 118) is essential to the development of well-structured programs.
- Division by zero is a logic error.
- To perform a floating-point calculation with integer values, cast (p. 124) one of the integers to type `double`.
- Java knows how to evaluate only arithmetic expressions in which the operands' types are identical. To ensure this, Java performs an operation called promotion (p. 124) on selected operands.
- The unary cast operator (p. 124) is formed by placing parentheses around the name of a type.

Section 4.11 Compound Assignment Operators

- The compound assignment operators (p. 130) abbreviate assignment expressions. Statements of the form

  variable `=` *variable operator expression*`;`

  where *operator* is one of the binary operators `+`, `-`, `*`, `/` or `%`, can be written in the form

  variable operator`=` *expression*`;`

- The `+=` operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator.

Section 4.12 Increment and Decrement Operators

- The unary increment operator, `++`, and the unary decrement operator, `--`, add 1 to or subtract 1 from the value of a numeric variable (p. 130).

- An increment or decrement operator that's prefixed (p. 131) to a variable is the prefix increment or prefix decrement operator, respectively. An increment or decrement operator that's postfixed (p. 131) to a variable is the postfix increment or postfix decrement operator, respectively.
- Using the prefix increment or decrement operator to add or subtract 1 is known as preincrementing or predecrementing, respectively.
- Preincrementing or predecrementing a variable causes the variable to be incremented or decremented by 1; then the new value of the variable is used in the expression in which it appears.
- Using the postfix increment or decrement operator to add or subtract 1 is known as postincrementing or postdecrementing, respectively.
- Postincrementing or postdecrementing the variable causes its value to be used in the expression in which it appears; then the variable's value is incremented or decremented by 1.
- When incrementing or decrementing a variable in a statement by itself, the prefix and postfix increment have the same effect, and the prefix and postfix decrement have the same effect.

Section 4.13 Primitive Types

- Java requires all variables to have a type. Thus, Java is referred to as a strongly typed language (p. 134).
- Java uses Unicode characters and IEEE 754 floating-point numbers.

Self-Review Exercises

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

a) All programs can be written in terms of three types of control structures: ________, ________ and ________.
b) The ________ statement is used to execute one action when a condition is true and another when that condition is false.
c) Repeating a set of instructions a specific number of times is called ________ repetition.
d) When it's not known in advance how many times a set of statements will be repeated, a(n) ________ value can be used to terminate the repetition.
e) The ________ structure is built into Java; by default, statements execute in the order they appear.
f) Instance variables of types `char`, `byte`, `short`, `int`, `long`, `float` and `double` are all given the value ________ by default.
g) Java is a(n) ________ language; it requires all variables to have a type.
h) If the increment operator is ________ to a variable, first the variable is incremented by 1, then its new value is used in the expression.

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

a) An algorithm is a procedure for solving a problem in terms of the actions to execute and the order in which they execute.
b) A set of statements contained within a pair of parentheses is called a block.
c) A selection statement specifies that an action is to be repeated while some condition remains true.
d) A nested control statement appears in the body of another control statement.
e) Java provides the arithmetic compound assignment operators `+=`, `-=`, `*=`, `/=` and `%=` for abbreviating assignment expressions.
f) The primitive types (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` and `double`) are portable across only Windows platforms.
g) Specifying the order in which statements execute in a program is called program control.
h) The unary cast operator `(double)` creates a temporary integer copy of its operand.

i) Instance variables of type `boolean` are given the value `true` by default.
j) Pseudocode helps you think out a program before attempting to write it in a programming language.

4.3 Write four different Java statements that each add 1 to integer variable `x`.

4.4 Write Java statements to accomplish each of the following tasks:
a) Use one statement to assign the sum of `x` and `y` to `z`, then increment `x` by 1.
b) Test whether variable `count` is greater than 10. If it is, print `"Count is greater than 10"`.
c) Use one statement to decrement the variable `x` by 1, then subtract it from variable `total` and store the result in variable `total`.
d) Calculate the remainder after `q` is divided by `divisor`, and assign the result to `q`. Write this statement in two different ways.

4.5 Write a Java statement to accomplish each of the following tasks:
a) Declare variables `sum` and `x` to be of type `int`.
b) Assign `1` to variable `x`.
c) Assign `0` to variable `sum`.
d) Add variable `x` to variable `sum`, and assign the result to variable `sum`.
e) Print `"The sum is: "`, followed by the value of variable `sum`.

4.6 Combine the statements that you wrote in Exercise 4.5 into a Java application that calculates and prints the sum of the integers from 1 to 10. Use a `while` statement to loop through the calculation and increment statements. The loop should terminate when the value of `x` becomes 11.

4.7 Determine the value of the variables in the statement `product *= x++;` after the calculation is performed. Assume that all variables are type `int` and initially have the value 5.

4.8 Identify and correct the errors in each of the following sets of code:

a)
```
while ( c <= 5 )
{
   product *= c;
   ++c;
```

b)
```
if ( gender == 1 )
   System.out.println( "Woman" );
else;
   System.out.println( "Man" );
```

4.9 What is wrong with the following `while` statement?

```
while ( z >= 0 )
   sum += z;
```

Answers to Self-Review Exercises

4.1 a) sequence, selection, repetition. b) `if…else`. c) counter-controlled (or definite). d) sentinel, signal, flag or dummy. e) sequence. f) `0` (zero). g) strongly typed. h) prefixed.

4.2 a) True. b) False. A set of statements contained within a pair of braces (`{` and `}`) is called a block. c) False. A repetition statement specifies that an action is to be repeated while some condition remains true. d) True. e) True. f) False. The primitive types (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` and `double`) are portable across all computer platforms that support Java. g) True. h) False. The unary cast operator `(double)` creates a temporary floating-point copy of its operand. i) False. Instance variables of type `boolean` are given the value `false` by default. j) True.

4.3
```
x = x + 1;
x += 1;
++x;
x++;
```

4.4 a) `z = x++ + y;`

b)
```
if ( count > 10 )
   System.out.println( "Count is greater than 10" );
```

c) `total -= --x;`

d) `q %= divisor;`
`q = q % divisor;`

4.5 a)
```
int sum;
int x;
```

b) `x = 1;`

c) `sum = 0;`

d) `sum += x;` or `sum = sum + x;`

e) `System.out.printf( "The sum is: %d\n", sum );`

4.6 The program is as follows:

```
// Exercise 4.6: Calculate.java
// Calculate the sum of the integers from 1 to 10
public class Calculate
{
   public static void main( String[] args )
   {
      int sum;
      int x;

      x = 1;   // initialize x to 1 for counting
      sum = 0; // initialize sum to 0 for totaling

      while ( x <= 10 ) // while x is less than or equal to 10
      {
         sum += x; // add x to sum
         ++x; // increment x
      } // end while

      System.out.printf( "The sum is: %d\n", sum );
   } // end main
} // end class Calculate
```

```
The sum is: 55
```

4.7 `product = 25, x = 6`

4.8 a) Error: The closing right brace of the `while` statement's body is missing.
Correction: Add a closing right brace after the statement `++c;`.

b) Error: The semicolon after `else` results in a logic error. The second output statement will always be executed.
Correction: Remove the semicolon after `else`.

4.9 The value of the variable `z` is never changed in the `while` statement. Therefore, if the loop-continuation condition `( z >= 0 )` is true, an infinite loop is created. To prevent an infinite loop from occurring, `z` must be decremented so that it eventually becomes less than 0.

Exercises

4.10 Compare and contrast the `if` single-selection statement and the `while` repetition statement. How are these two statements similar? How are they different?

4.11 Explain what happens when a Java program attempts to divide one integer by another. What happens to the fractional part of the calculation? How can you avoid that outcome?

4.12 Describe the two ways in which control statements can be combined.

4.13 What type of repetition would be appropriate for calculating the sum of the first 100 positive integers? What type would be appropriate for calculating the sum of an arbitrary number of positive integers? Briefly describe how each of these tasks could be performed.

4.14 What is the difference between preincrementing and postincrementing a variable?

4.15 Identify and correct the errors in each of the following pieces of code. [*Note:* There may be more than one error in each piece of code.]

a)
```
if ( age >= 65 );
   System.out.println( "Age is greater than or equal to 65" );
else
   System.out.println( "Age is less than 65 )";
```

b)
```
int x = 1, total;
while ( x <= 10 )
{
   total += x;
   ++x;
}
```

c)
```
while ( x <= 100 )
   total += x;
   ++x;
```

d)
```
while ( y > 0 )
{
   System.out.println( y );
   ++y;
```

4.16 What does the following program print?

```
// Exercise 4.16: Mystery.java
public class Mystery
{
   public static void main( String[] args )
   {
      int y;
      int x = 1;
      int total = 0;

      while ( x <= 10 )
      {
         y = x * x;
         System.out.println( y );
         total += y;
         ++x;
      } // end while

      System.out.printf( "Total is %d\n", total );
   } // end main
} // end class Mystery
```

For Exercise 4.17 through Exercise 4.20, perform each of the following steps:

a) Read the problem statement.

b) Formulate the algorithm using pseudocode and top-down, stepwise refinement.

c) Write a Java program.
d) Test, debug and execute the Java program.
e) Process three complete sets of data.

4.17 *(Gas Mileage)* Drivers are concerned with the mileage their automobiles get. One driver has kept track of several trips by recording the miles driven and gallons used for each tankful. Develop a Java application that will input the miles driven and gallons used (both as integers) for each trip. The program should calculate and display the miles per gallon obtained for each trip and print the combined miles per gallon obtained for all trips up to this point. All averaging calculations should produce floating-point results. Use class `Scanner` and sentinel-controlled repetition to obtain the data from the user.

4.18 *(Credit Limit Calculator)* Develop a Java application that determines whether any of several department-store customers has exceeded the credit limit on a charge account. For each customer, the following facts are available:

a) account number
b) balance at the beginning of the month
c) total of all items charged by the customer this month
d) total of all credits applied to the customer's account this month
e) allowed credit limit.

The program should input all these facts as integers, calculate the new balance (= *beginning balance + charges – credits*), display the new balance and determine whether the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the message `"Credit limit exceeded"`.

4.19 *(Sales Commission Calculator)* A large company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9% of their gross sales for that week. For example, a salesperson who sells $5000 worth of merchandise in a week receives $200 plus 9% of $5000, or a total of $650. You've been supplied with a list of the items sold by each salesperson. The values of these items are as follows:

```
Item     Value
1        239.99
2        129.75
3         99.95
4        350.89
```

Develop a Java application that inputs one salesperson's items sold for last week and calculates and displays that salesperson's earnings. There's no limit to the number of items that can be sold.

4.20 *(Salary Calculator)* Develop a Java application that determines the gross pay for each of three employees. The company pays straight time for the first 40 hours worked by each employee and time and a half for all hours worked in excess of 40. You're given a list of the employees, their number of hours worked last week and their hourly rates. Your program should input this information for each employee, then determine and display the employee's gross pay. Use class `Scanner` to input the data.

4.21 *(Find the Largest Number)* The process of finding the largest value is used frequently in computer applications. For example, a program that determines the winner of a sales contest would input the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a pseudocode program, then a Java application that inputs a series of 10 integers and determines and prints the largest integer. Your program should use at least the following three variables:

a) `counter`: A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed).
b) `number`: The integer most recently input by the user.
c) `largest`: The largest number found so far.

4.22 *(Tabular Output)* Write a Java application that uses looping to print the following table of values:

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

4.23 *(Find the Two Largest Numbers)* Using an approach similar to that for Exercise 4.21, find the *two* largest values of the 10 values entered. [*Note:* You may input each number only once.]

4.24 *(Validating User Input)* Modify the program in Fig. 4.12 to validate its inputs. For any input, if the value entered is other than 1 or 2, keep looping until the user enters a correct value.

4.25 What does the following program print?

```java
// Exercise 4.25: Mystery2.java
public class Mystery2
{
   public static void main( String[] args )
   {
      int count = 1;

      while ( count <= 10 )
      {
         System.out.println( count % 2 == 1 ? "****" : "++++++++" );
         ++count;
      } // end while
   } // end main
} // end class Mystery2
```

4.26 What does the following program print?

```java
// Exercise 4.26: Mystery3.java
public class Mystery3
{
   public static void main( String[] args )
   {
      int row = 10;
      int column;

      while ( row >= 1 )
      {
         column = 1;

         while ( column <= 10 )
         {
            System.out.print( row % 2 == 1 ? "<" : ">" );
            ++column;
         } // end while

         --row;
         System.out.println();
      } // end while
   } // end main
} // end class Mystery3
```

4.27 *(Dangling-`else` Problem)* Determine the output for each of the given sets of code when `x` is `9` and `y` is `11` and when `x` is `11` and `y` is `9`. The compiler ignores the indentation in a Java program. Also, the Java compiler always associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{}`). On first glance, you may not be sure which `if` a particular `else` matches—this situation is referred to as the "dangling-`else` problem." We've eliminated the indentation from the following code to make the problem more challenging. [*Hint:* Apply the indentation conventions you've learned.]

a)
```
if ( x < 10 )
if ( y > 10 )
System.out.println( "*****" );
else
System.out.println( "#####" );
System.out.println( "$$$$$" );
```

b)
```
if ( x < 10 )
{
if ( y > 10 )
System.out.println( "*****"  );
}
else
{
System.out.println( "#####" );
System.out.println( "$$$$$" );
}
```

4.28 *(Another Dangling-`else` Problem)* Modify the given code to produce the output shown in each part of the problem. Use proper indentation techniques. Make no changes other than inserting braces and changing the indentation of the code. The compiler ignores indentation in a Java program. We've eliminated the indentation from the given code to make the problem more challenging. [*Note:* It's possible that no modification is necessary for some of the parts.]

```
if ( y == 8 )
if ( x == 5 )
System.out.println( "@@@@@" );
else
System.out.println( "#####" );
System.out.println( "$$$$$" );
System.out.println( "&&&&&" );
```

a) Assuming that `x = 5` and `y = 8`, the following output is produced:

```
@@@@@
$$$$$
&&&&&
```

b) Assuming that `x = 5` and `y = 8`, the following output is produced:

```
@@@@@
```

c) Assuming that `x = 5` and `y = 8`, the following output is produced:

```
@@@@@
```

d) Assuming that `x = 5` and `y = 7`, the following output is produced. [*Note:* The last three output statements after the `else` are all part of a block.]

```
#####
$$$$$
&&&&&
```

4.29 *(Square of Asterisks)* Write an application that prompts the user to enter the size of the side of a square, then displays a hollow square of that size made of asterisks. Your program should work for squares of all side lengths between 1 and 20.

4.30 *(Palindromes)* A palindrome is a sequence of characters that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write an application that reads in a five-digit integer and determines whether it's a palindrome. If the number is not five digits long, display an error message and allow the user to enter a new value.

4.31 *(Printing the Decimal Equivalent of a Binary Number)* Write an application that inputs an integer containing only 0s and 1s (i.e., a binary integer) and prints its decimal equivalent. [*Hint:* Use the remainder and division operators to pick off the binary number's digits one at a time, from right to left. In the decimal number system, the rightmost digit has a positional value of 1 and the next digit to the left a positional value of 10, then 100, then 1000, and so on. The decimal number 234 can be interpreted as 4 * 1 + 3 * 10 + 2 * 100. In the binary number system, the rightmost digit has a positional value of 1, the next digit to the left a positional value of 2, then 4, then 8, and so on. The decimal equivalent of binary 1101 is 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8, or 1 + 0 + 4 + 8 or, 13.]

4.32 *(Checkerboard Pattern of Asterisks)* Write an application that uses only the output statements

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

to display the checkerboard pattern that follows. A `System.out.println` method call with no arguments causes the program to output a single newline character. [*Hint:* Repetition statements are required.]

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

4.33 *(Multiples of 2 with an Infinite Loop)* Write an application that keeps displaying in the command window the multiples of the integer 2—namely, 2, 4, 8, 16, 32, 64, and so on. Your loop should not terminate (i.e., it should create an infinite loop). What happens when you run this program?

4.34 *(What's Wrong with This Code?)* What is wrong with the following statement? Provide the correct statement to add one to the sum of `x` and `y`.

```
System.out.println( ++(x + y) );
```

4.35 *(Sides of a Triangle)* Write an application that reads three nonzero values entered by the user and determines and prints whether they could represent the sides of a triangle.

4.36 *(Sides of a Right Triangle)* Write an application that reads three nonzero integers and determines and prints whether they could represent the sides of a right triangle.

4.37 *(Factorial)* The factorial of a nonnegative integer *n* is written as *n*! (pronounced "*n* factorial") and is defined as follows:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 \quad \text{(for values of } n \text{ greater than or equal to 1)}$$

and

$n! = 1$ (for $n = 0$)

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is 120.

a) Write an application that reads a nonnegative integer and computes and prints its factorial.

b) Write an application that estimates the value of the mathematical constant *e* by using the following formula. Allow the user to enter the number of terms to calculate.

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) Write an application that computes the value of e^x by using the following formula. Allow the user to enter the number of terms to calculate.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Making a Difference

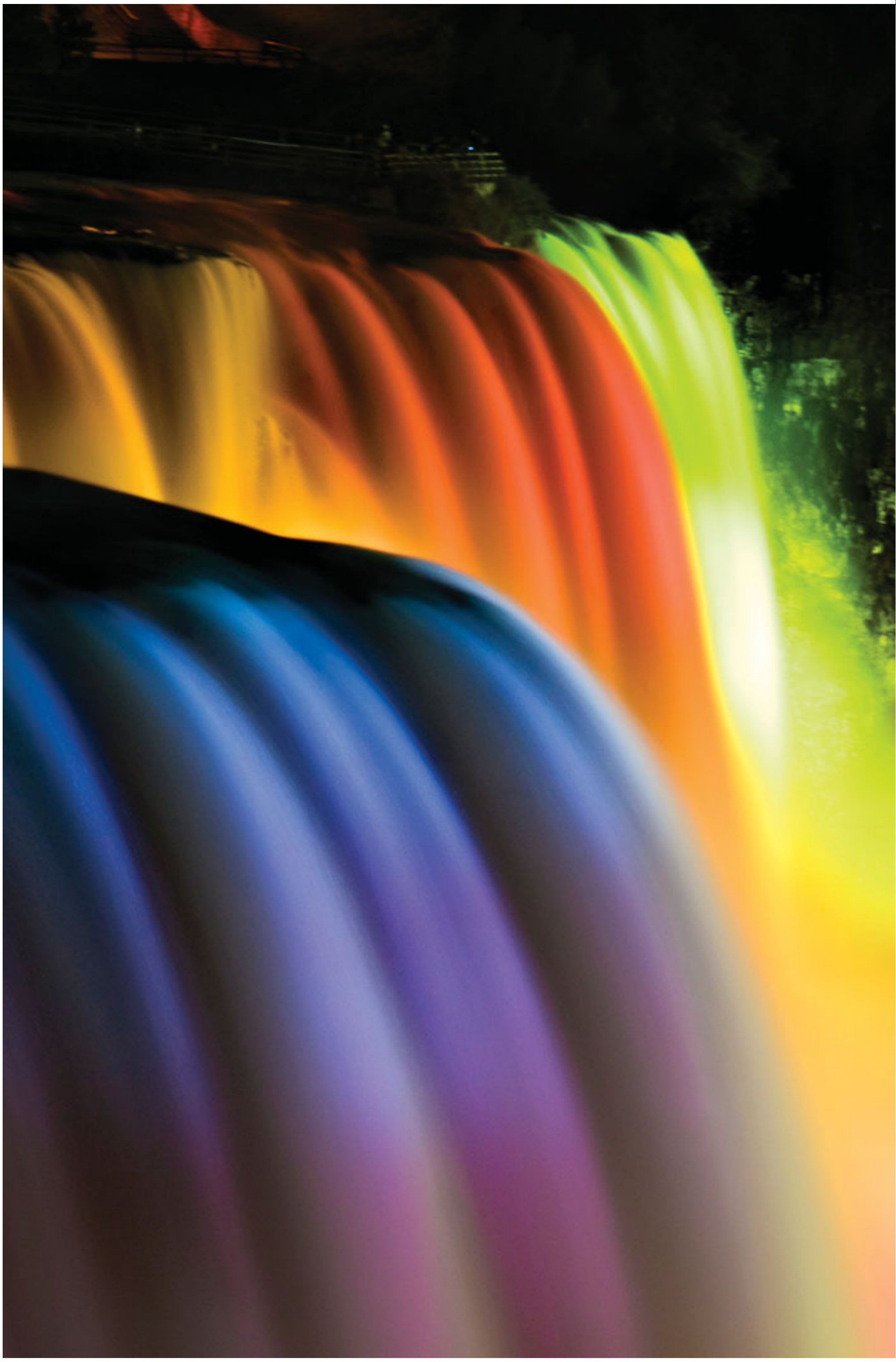
4.38 *(Enforcing Privacy with Cryptography)* The explosive growth of Internet communications and data storage on Internet-connected computers has greatly increased privacy concerns. The field of cryptography is concerned with coding data to make it difficult (and hopefully—with the most advanced schemes—impossible) for unauthorized users to read. In this exercise you'll investigate a simple scheme for encrypting and decrypting data. A company that wants to send data over the Internet has asked you to write a program that will encrypt it so that it may be transmitted more securely. All the data is transmitted as four-digit integers. Your application should read a four-digit integer entered by the user and encrypt it as follows: Replace each digit with the result of adding 7 to the digit and getting the remainder after dividing the new value by 10. Then swap the first digit with the third, and swap the second digit with the fourth. Then print the encrypted integer. Write a separate application that inputs an encrypted four-digit integer and decrypts it (by reversing the encryption scheme) to form the original number. [*Optional reading project:* Research "public key cryptography" in general and the PGP (Pretty Good Privacy) specific public key scheme. You may also want to investigate the RSA scheme, which is widely used in industrial-strength applications.]

4.39 *(World Population Growth)* World population has grown considerably over the centuries. Continued growth could eventually challenge the limits of breathable air, drinkable water, arable cropland and other limited resources. There's evidence that growth has been slowing in recent years and that world population could peak some time this century, then start to decline.

For this exercise, research world population growth issues online. *Be sure to investigate various viewpoints.* Get estimates for the current world population and its growth rate (the percentage by which it's likely to increase this year). Write a program that calculates world population growth each year for the next 75 years, *using the simplifying assumption that the current growth rate will stay constant*. Print the results in a table. The first column should display the year from year 1 to year 75. The second column should display the anticipated world population at the end of that year. The third column should display the numerical increase in the world population that would occur that year. Using your results, determine the year in which the population would be double what it is today, if this year's growth rate were to persist.

Control Statements: Part 2

5



The wheel is come full circle.
—William Shakespeare

—Robert Frost

All the evolution we know of proceeds from the vague to the definite.
—Charles Sanders Peirce

Objectives

In this chapter you'll learn:

- The essentials of counter-controlled repetition.
- To use the `for` and `do`...`while` repetition statements to execute statements in a program repeatedly.
- To understand multiple selection using the `switch` selection statement.
- To use the `break` and `continue` program control statements to alter the flow of control.
- To use the logical operators to form complex conditional expressions in control statements.

Outline

5.1 Introduction

This chapter continues our presentation of structured programming theory and principles by introducing all but one of Java's remaining control statements. We demonstrate Java's `for`, `do…while` and `switch` statements. Through a series of short examples using `while` and `for`, we explore the essentials of counter-controlled repetition. We create a version of class `GradeBook` that uses a `switch` statement to count the number of A, B, C, D and F grade equivalents in a set of numeric grades entered by the user. We introduce the `break` and `continue` program-control statements. We discuss Java's logical operators, which enable you to use more complex conditional expressions in control statements. Finally, we summarize Java's control statements and the proven problem-solving techniques presented in this chapter and Chapter 4.

5.2 Essentials of Counter-Controlled Repetition

This section uses the `while` repetition statement introduced in Chapter 4 to formalize the elements required to perform counter-controlled repetition, which requires

1. a **control variable** (or loop counter)
2. the **initial value** of the control variable
3. the **increment** (or **decrement**) by which the control variable is modified each time through the loop (also known as **each iteration of the loop**)
4. the **loop-continuation condition** that determines if looping should continue.

To see these elements of counter-controlled repetition, consider the application of Fig. 5.1, which uses a loop to display the numbers from 1 through 10.

```
// Fig. 5.1: WhileCounter.java
// Counter-controlled repetition with the while repetition statement.

public class WhileCounter
{
   public static void main( String[] args )
   {
```

Fig. 5.1 | Counter-controlled repetition with the `while` repetition statement. (Part 1 of 2.)

```
8        int counter = 1; // declare and initialize control variable
9
10       while ( counter <= 10 ) // loop-continuation condition
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter; // increment control variable by 1
14       } // end while
15
16       System.out.println(); // output a newline
17    } // end main
18 } // end class WhileCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.1 | Counter-controlled repetition with the `while` repetition statement. (Part 2 of 2.)

In Fig. 5.1, the elements of counter-controlled repetition are defined in lines 8, 10 and 13. Line 8 declares the control variable (`counter`) as an `int`, reserves space for it in memory and sets its initial value to `1`. Variable `counter` also could have been declared and initialized with the following local-variable declaration and assignment statements:

```
int counter; // declare counter
counter = 1; // initialize counter to 1
```

Line 12 displays control variable `counter`'s value during each iteration of the loop. Line 13 increments the control variable by 1 for each iteration of the loop. The loop-continuation condition in the `while` (line 10) tests whether the value of the control variable is less than or equal to `10` (the final value for which the condition is `true`). The program performs the body of this `while` even when the control variable is `10`. The loop terminates when the control variable exceeds `10` (i.e., `counter` becomes `11`).

Common Programming Error 5.1

Because floating-point values may be approximate, controlling loops with floating-point variables may result in imprecise counter values and inaccurate termination tests.

Error-Prevention Tip 5.1

Use integers to control counting loops.

The program in Fig. 5.1 can be made more concise by initializing `counter` to `0` in line 8 and preincrementing `counter` in the `while` condition as follows:

```
while ( ++counter <= 10 ) // loop-continuation condition
   System.out.printf( "%d  ", counter );
```

This code saves a statement (and eliminates the need for braces around the loop's body), because the `while` condition performs the increment before testing the condition. (Recall from Section 4.12 that the precedence of `++` is higher than that of `<=`.) Coding in such a condensed fashion takes practice, might make code more difficult to read, debug, modify and maintain, and typically should be avoided.

Software Engineering Observation 5.1

"Keep it simple" is good advice for most of the code you'll write.

5.3 for Repetition Statement

Section 5.2 presented the essentials of counter-controlled repetition. The `while` statement can be used to implement any counter-controlled loop. Java also provides the **for repetition statement**, which specifies the counter-controlled-repetition details in a single line of code. Figure 5.2 reimplements the application of Fig. 5.1 using `for`.

```
1  // Fig. 5.2: ForCounter.java
2  // Counter-controlled repetition with the for repetition statement.
3
4  public class ForCounter
5  {
6     public static void main( String[] args )
7     {
8        // for statement header includes initialization,
9        // loop-continuation condition and increment
10       for ( int counter = 1; counter <= 10; counter++ )
11          System.out.printf( "%d  ", counter );
12
13       System.out.println(); // output a newline
14    } // end main
15 } // end class ForCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.2 | Counter-controlled repetition with the `for` repetition statement.

When the `for` statement (lines 10–11) begins executing, the control variable `counter` is declared and initialized to `1`. (Recall from Section 5.2 that the first two elements of counter-controlled repetition are the control variable and its initial value.) Next, the program checks the loop-continuation condition, `counter <= 10`, which is between the two required semicolons. Because the initial value of `counter` is `1`, the condition initially is true. Therefore, the body statement (line 11) displays control variable `counter`'s value, namely `1`. After executing the loop's body, the program increments `counter` in the expression `counter++`, which appears to the right of the second semicolon. Then the loop-continuation test is performed again to determine whether the program should continue with the next iteration of the loop. At this point, the control variable's value is `2`, so the condition is still true (the final value is not exceeded)—thus, the program performs the body statement again (i.e., the next iteration of the loop). This process continues until the numbers 1 through 10 have been displayed and the `counter`'s value becomes `11`, causing the loop-continuation test to fail and repetition to terminate (after 10 repetitions of the loop body). Then the program performs the first statement after the `for`—in this case, line 13.

Figure 5.2 uses (in line 10) the loop-continuation condition `counter <= 10`. If you incorrectly specified `counter < 10` as the condition, the loop would iterate only nine times. This is a common logic error called an **off-by-one error**.

Common Programming Error 5.2

Using an incorrect relational operator or an incorrect final value of a loop counter in the loop-continuation condition of a repetition statement can cause an off-by-one error.

Error-Prevention Tip 5.2

Using the final value in the condition of a `while` *or* `for` *statement and using the* `<=` *relational operator helps avoid off-by-one errors. For a loop that prints the values 1 to 10, the loop-continuation condition should be* `counter <= 10` *rather than* `counter < 10` *(which causes an off-by-one error) or* `counter < 11` *(which is correct). Many programmers prefer so-called zero-based counting, in which to count 10 times,* `counter` *would be initialized to zero and the loop-continuation test would be* `counter < 10`*.*

A Closer Look at the `for` *Statement's Header*

Figure 5.3 takes a closer look at the `for` statement in Fig. 5.2. The `for`'s first line (including the keyword `for` and everything in parentheses after `for`)—line 10 in Fig. 5.2—is sometimes called the **`for` statement header**. The `for` header "does it all"—it specifies each item needed for counter-controlled repetition with a control variable. If there's more than one statement in the body of the `for`, braces are required to define the body of the loop.

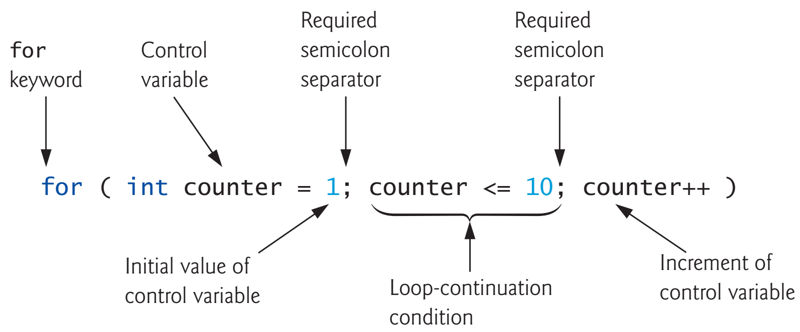


Fig. 5.3 | `for` statement header components.

General Format of a `for` *Statement*

The general format of the `for` statement is

```
for ( initialization; loopContinuationCondition; increment )
   statement
```

where the *initialization* expression names the loop's control variable and optionally provides its initial value, *loopContinuationCondition* determines whether the loop should continue executing and *increment* modifies the control variable's value (possibly an increment or decrement), so that the loop-continuation condition eventually becomes false. The two semicolons in the `for` header are required. If the loop-continuation condition is initially `false`, the program does *not* execute the `for` statement's body. Instead, execution proceeds with the statement following the `for`.

Representing a `for` *Statement with an Equivalent* `while` *Statement*

In most cases, the `for` statement can be represented with an equivalent `while` statement as follows:

```
initialization;
while ( loopContinuationCondition )
{
   statement
   increment;
}
```

In Section 5.7, we show a case in which a `for` statement cannot be represented with an equivalent `while` statement.

Typically, `for` statements are used for counter-controlled repetition and `while` statements for sentinel-controlled repetition. However, `while` and `for` can each be used for either repetition type.

Scope of a `for` *Statement's Control Variable*

If the *initialization* expression in the `for` header declares the control variable (i.e., the control variable's type is specified before the variable name, as in Fig. 5.2), the control variable can be used *only* in that `for` statement—it will not exist outside it. This restricted use is known as the variable's **scope**. The scope of a variable defines where it can be used in a program. For example, a local variable can be used *only* in the method that declares it and *only* from the point of declaration through the end of the method. Scope is discussed in detail in Chapter 6, Methods: A Deeper Look.

Common Programming Error 5.3

When a `for` statement's control variable is declared in the initialization section of the `for`'s header, using the control variable after the `for`'s body is a compilation error.

Expressions in a `for` *Statement's Header Are Optional*

All three expressions in a `for` header are optional. If the *loopContinuationCondition* is omitted, Java assumes that the loop-continuation condition is always true, thus creating an infinite loop. You might omit the *initialization* expression if the program initializes the control variable before the loop. You might omit the *increment* expression if the program calculates the increment with statements in the loop's body or if no increment is needed. The increment expression in a `for` acts as if it were a standalone statement at the end of the `for`'s body. Therefore, the expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

are equivalent increment expressions in a `for` statement. Many programmers prefer `counter++` because it's concise and because a `for` loop evaluates its increment expression *after* its body executes, so the postfix increment form seems more natural. In this case, the variable being incremented does not appear in a larger expression, so preincrementing and postincrementing actually have the same effect.

Common Programming Error 5.4

Placing a semicolon immediately to the right of the right parenthesis of a `for` header makes that `for`'s body an empty statement. This is normally a logic error.

Error-Prevention Tip 5.3

Infinite loops occur when the loop-continuation condition in a repetition statement never becomes `false`*. To prevent this situation in a counter-controlled loop, ensure that the control variable is incremented (or decremented) during each iteration of the loop. In a sentinel-controlled loop, ensure that the sentinel value is able to be input.*

Placing Arithmetic Expressions in a `for` *Statement's Header*

The initialization, loop-continuation condition and increment portions of a `for` statement can contain arithmetic expressions. For example, assume that `x = 2` and `y = 10`. If `x` and `y` are not modified in the body of the loop, the statement

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

is equivalent to the statement

```
for ( int j = 2; j <= 80; j += 5 )
```

The increment of a `for` statement may also be *negative*, in which case it's really a *decrement*, and the loop counts *downward*.

Using a `for` *Statement's Control Variable in the Statements's Body*

Programs frequently display the control-variable value or use it in calculations in the loop body, but this use is not required. The control variable is commonly used to control repetition without being mentioned in the body of the `for`.

Error-Prevention Tip 5.4

Although the value of the control variable can be changed in the body of a `for` *loop, avoid doing so, because this practice can lead to subtle errors.*

UML Activity Diagram for the `for` *Statement*

The `for` statement's UML activity diagram is similar to that of the `while` statement (Fig. 4.4). Figure 5.4 shows the activity diagram of the `for` statement in Fig. 5.2. The

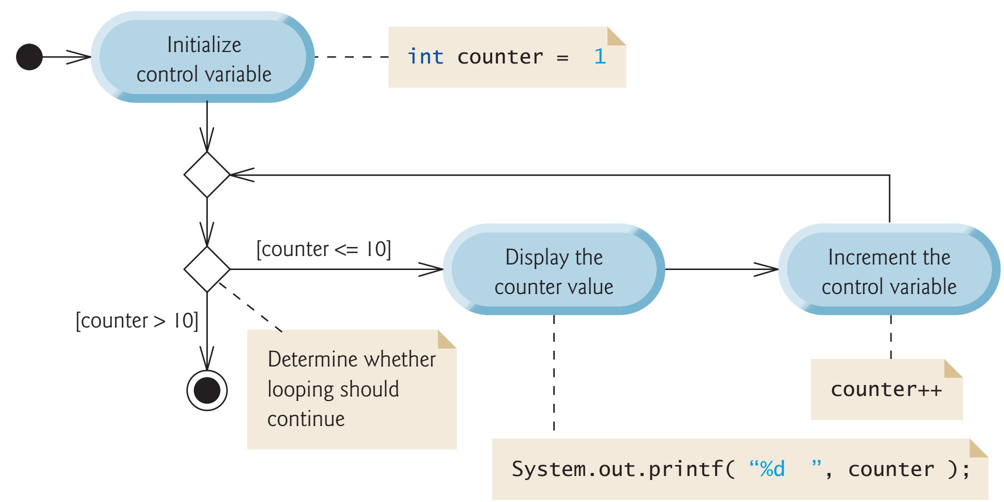


Fig. 5.4 | UML activity diagram for the `for` statement in Fig. 5.2.

diagram makes it clear that initialization occurs *once before* the loop-continuation test is evaluated the first time, and that incrementing occurs *each* time through the loop *after* the body statement executes.

5.4 Examples Using the `for` Statement

The following examples show techniques for varying the control variable in a `for` statement. In each case, we write the appropriate `for` header. Note the change in the relational operator for loops that decrement the control variable.

a) Vary the control variable from `1` to `100` in increments of `1`.

```
for ( int i = 1; i <= 100; i++ )
```

b) Vary the control variable from `100` to `1` in decrements of `1`.

```
for ( int i = 100; i >= 1; i-- )
```

c) Vary the control variable from `7` to `77` in increments of `7`.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Vary the control variable from `20` to `2` in decrements of `2`.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Vary the control variable over the values `2, 5, 8, 11, 14, 17, 20`.

```
for ( int i = 2; i <= 20; i += 3 )
```

f) Vary the control variable over the values `99, 88, 77, 66, 55, 44, 33, 22, 11, 0`.

```
for ( int i = 99; i >= 0; i -= 11 )
```

Common Programming Error 5.5

Using an incorrect relational operator in the loop-continuation condition of a loop that counts downward (e.g., using `i <= 1` instead of `i >= 1` in a loop counting down to 1) is usually a logic error.

Application: Summing the Even Integers from 2 to 20

We now consider two sample applications that demonstrate simple uses of `for`. The application in Fig. 5.5 uses a `for` statement to sum the even integers from 2 to 20 and store the result in an `int` variable called `total`.

```
// Fig. 5.5: Sum.java
// Summing integers with the for statement.

public class Sum
{
   public static void main( String[] args )
   {
      int total = 0; // initialize total

```

Fig. 5.5 | Summing integers with the `for` statement. (Part 1 of 2.)

```
10          // total even integers from 2 through 20
11          for ( int number = 2; number <= 20; number += 2 )
12             total += number;
13
14          System.out.printf( "Sum is %d\n", total ); // display results
15       } // end main
16    } // end class Sum
```

```
Sum is 110
```

Fig. 5.5 | Summing integers with the `for` statement. (Part 2 of 2.)

The *initialization* and *increment* expressions can be comma-separated lists that enable you to use multiple initialization expressions or multiple increment expressions. For example, *although this is discouraged*, you could merge the body of the `for` statement in lines 11–12 of Fig. 5.5 into the increment portion of the `for` header by using a comma as follows:

```
for ( int number = 2; number <= 20; total += number, number += 2 )
   ; // empty statement
```

Good Programming Practice 5.1

For readability limit the size of control-statement headers to a single line if possible.

Application: Compound-Interest Calculations

Let's use the `for` statement to compute compound interest. Consider the following problem:

> *A person invests $1000 in a savings account yielding 5% interest. Assuming that all the interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:*
>
> $$a = p\,(1 + r)^n$$
>
> *where*
>
> p is the original amount invested (i.e., the principal)
> r is the annual interest rate (e.g., use 0.05 for 5%)
> n is the number of years
> a is the amount on deposit at the end of the nth year.

The solution to this problem (Fig. 5.6) involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit. Lines 8–10 in method `main` declare `double` variables `amount`, `principal` and `rate`, and initialize `principal` to `1000.0` and `rate` to `0.05`. Java treats floating-point constants like `1000.0` and `0.05` as type `double`. Similarly, Java treats whole-number constants like `7` and `-22` as type `int`.

```
1  // Fig. 5.6: Interest.java
2  // Compound-interest calculations with for.
3
4  public class Interest
5  {
```

Fig. 5.6 | Compound-interest calculations with `for`. (Part 1 of 2.)

```
6      public static void main( String[] args )
7      {
8         double amount; // amount on deposit at end of each year
9         double principal = 1000.0; // initial amount before interest
10        double rate = 0.05; // interest rate
11
12        // display headers
13        System.out.printf( "%s%20s\n", "Year", "Amount on deposit" );
14
15        // calculate amount on deposit for each of ten years
16        for ( int year = 1; year <= 10; year++ )
17        {
18           // calculate new amount for specified year
19           amount = principal * Math.pow( 1.0 + rate, year );
20
21           // display the year and the amount
22           System.out.printf( "%4d%,20.2f\n", year, amount );
23        } // end for
24     } // end main
25  } // end class Interest
```

```
Year   Amount on deposit
   1            1,050.00
   2            1,102.50
   3            1,157.63
   4            1,215.51
   5            1,276.28
   6            1,340.10
   7            1,407.10
   8            1,477.46
   9            1,551.33
  10            1,628.89
```

Fig. 5.6 | Compound-interest calculations with `for`. (Part 2 of 2.)

Formatting Strings with Field Widths and Justification

Line 13 outputs the headers for two columns of output. The first column displays the year and the second column the amount on deposit at the end of that year. We use the format specifier `%20s` to output the `String` `"Amount on Deposit"`. The integer `20` between the `%` and the conversion character `s` indicates that the value should be displayed with a **field width** of 20—that is, `printf` displays the value with at least 20 character positions. If the value to be output is less than 20 character positions wide (17 characters in this example), the value is **right justified** in the field by default. If the `year` value to be output were more than four character positions wide, the field width would be extended to the right to accommodate the entire value—this would push the `amount` field to the right, upsetting the neat columns of our tabular output. To output values **left justified**, simply precede the field width with the **minus sign (–) formatting flag** (e.g., `%-20s`).

Performing the Interest Calculations

The `for` statement (lines 16–23) executes its body 10 times, varying control variable `year` from 1 to 10 in increments of 1. This loop terminates when `year` becomes 11. (Variable `year` represents *n* in the problem statement.)

Classes provide methods that perform common tasks on objects. In fact, most methods must be called on a specific object. For example, to output text in Fig. 5.6, line 13 calls method `printf` on the `System.out` object. Many classes also provide methods that perform common tasks and do *not* require objects. These are called `static` methods. For example, Java does not include an exponentiation operator, so the designers of Java's `Math` class defined `static` method `pow` for raising a value to a power. You can call a `static` method by specifying the class name followed by a dot (`.`) and the method name, as in

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

In Chapter 6, you'll learn how to implement `static` methods in your own classes.

We use `static` method **`pow`** of class **`Math`** to perform the compound-interest calculation in Fig. 5.6. `Math.pow(`x`, `y`)` calculates the value of x raised to the y^{th} power. The method receives two `double` arguments and returns a `double` value. Line 19 performs the calculation $a = p(1 + r)^n$, where a is `amount`, p is `principal`, r is `rate` and n is `year`. Class `Math` is defined in package `java.lang`, so you do *not* need to import class `Math` to use it.

The body of the `for` statement contains the calculation `1.0 + rate`, which appears as an argument to the `Math.pow` method. In fact, this calculation produces the same result each time through the loop, so repeating it every iteration of the loop is wasteful.

Performance Tip 5.1

In loops, avoid calculations for which the result never changes—such calculations should typically be placed before the loop. Many of today's sophisticated optimizing compilers will place such calculations outside loops in the compiled code.

Formatting Floating-Point Numbers

After each calculation, line 22 outputs the year and the amount on deposit at the end of that year. The year is output in a field width of four characters (as specified by `%4d`). The amount is output as a floating-point number with the format specifier `%,20.2f`. The **comma (`,`) formatting flag** indicates that the floating-point value should be output with a **grouping separator**. The actual separator used is specific to the user's locale (i.e., country). For example, in the United States, the number will be output using commas to separate every three digits and a decimal point to separate the fractional part of the number, as in 1,234.45. The number `20` in the format specification indicates that the value should be output right justified in a field width of 20 characters. The `.2` specifies the formatted number's precision—in this case, the number is rounded to the nearest hundredth and output with two digits to the right of the decimal point.

A Warning about Displaying Rounded Values

We declared variables `amount`, `principal` and `rate` to be of type `double` in this example. We're dealing with fractional parts of dollars and thus need a type that allows decimal points in its values. Unfortunately, floating-point numbers can cause trouble. Here's a simple explanation of what can go wrong when using `double` (or `float`) to represent dollar amounts (assuming that dollar amounts are displayed with two digits to the right of the decimal point): Two `double` dollar amounts stored in the machine could be 14.234 (which would normally be rounded to 14.23 for display purposes) and 18.673 (which would normally be rounded to 18.67 for display purposes). When these amounts are added, they produce the internal sum 32.907, which would normally be rounded to 32.91 for display purposes. Thus, your output could appear as

```
   14.23
 + 18.67
 -------
   32.91
```

but a person adding the individual numbers as displayed would expect the sum to be 32.90. You've been warned!

Error-Prevention Tip 5.5

Do not use variables of type `double` *(or* `float`*) to perform precise monetary calculations. The imprecision of floating-point numbers can cause errors. In the exercises, you'll learn how to use integers to perform precise monetary calculations. Java also provides class* `java.math.BigDecimal` *to perform precise monetary calculations. For more information, see* `download.oracle.com/javase/6/docs/api/java/math//BigDecimal.html`.

5.5 do…while Repetition Statement

The **do…while repetition statement** is similar to the `while` statement. In the `while`, the program tests the loop-continuation condition at the beginning of the loop, before executing the loop's body; if the condition is false, the body *never* executes. The `do…while` statement tests the loop-continuation condition *after* executing the loop's body; therefore, *the body always executes at least once*. When a `do…while` statement terminates, execution continues with the next statement in sequence. Figure 5.7 uses a `do…while` (lines 10–14) to output the numbers 1–10.

```
1  // Fig. 5.7: DoWhileTest.java
2  // do...while repetition statement.
3
4  public class DoWhileTest
5  {
6     public static void main( String[] args )
7     {
8        int counter = 1; // initialize counter
9
10       do
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter;
14       } while ( counter <= 10 ); // end do...while
15
16       System.out.println(); // outputs a newline
17    } // end main
18 } // end class DoWhileTest
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.7 | `do…while` repetition statement.

Line 8 declares and initializes control variable `counter`. Upon entering the `do…while` statement, line 12 outputs `counter`'s value and line 13 increments `counter`. Then the program evaluates the loop-continuation test at the *bottom* of the loop (line 14). If the condition is true, the loop continues from the first body statement (line 12). If the condition is false, the loop terminates and the program continues with the next statement after the loop.

Figure 5.8 contains the UML activity diagram for the `do…while` statement. This diagram makes it clear that the loop-continuation condition is not evaluated until *after* the loop performs the action state at least once. Compare this activity diagram with that of the `while` statement (Fig. 4.4).

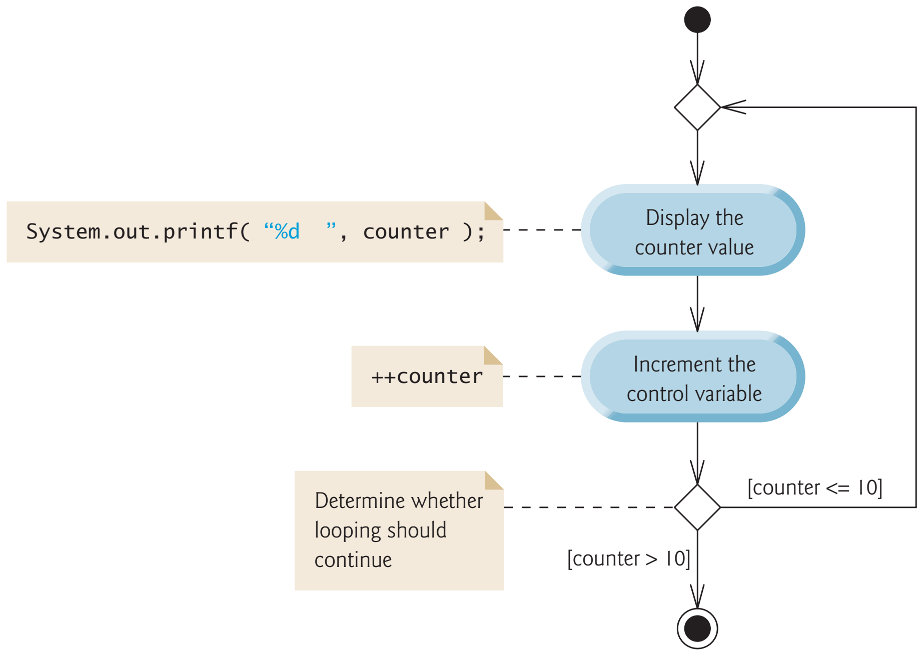


Fig. 5.8 | `do...while` repetition statement UML activity diagram.

It isn't necessary to use braces in the `do…while` repetition statement if there's only one statement in the body. However, many programmers include the braces, to avoid confusion between the `while` and `do…while` statements. For example,

```
while ( condition )
```

is normally the first line of a `while` statement. A `do…while` statement with no braces around a single-statement body appears as:

```
do
   statement
while ( condition );
```

which can be confusing. A reader may misinterpret the last line—`while(` *condition* `);`—as a `while` statement containing an empty statement (the semicolon by itself). Thus, the `do…while` statement with one body statement is usually written as follows:

```
do
{
   statement
} while ( condition );
```

Good Programming Practice 5.2

Always include braces in a `do…while` *statement. This helps eliminate ambiguity between the* `while` *statement and a* `do…while` *statement containing only one statement.*

5.6 switch Multiple-Selection Statement

Chapter 4 discussed the `if` single-selection statement and the `if…else` double-selection statement. The **`switch` multiple-selection statement** performs different actions based on the possible values of a **constant integral expression** of type `byte`, `short`, `int` or `char`.

GradeBook Class with switch Statement to Count A, B, C, D and F Grades

Figure 5.9 enhances class `GradeBook` from Chapters 3–4. The new version we now present not only calculates the average of a set of numeric grades entered by the user, but uses a `switch` statement to determine whether each grade is the equivalent of an A, B, C, D or F and to increment the appropriate grade counter. The class also displays a summary of the number of students who received each grade. Refer to Fig. 5.10 for sample inputs and outputs of the `GradeBookTest` application that uses class `GradeBook` to process a set of grades.

```java
// Fig. 5.9: GradeBook.java
// GradeBook class uses switch statement to count letter grades.
import java.util.Scanner; // program uses class Scanner

public class GradeBook
{
   private String courseName; // name of course this GradeBook represents
   // int instance variables are initialized to 0 by default
   private int total; // sum of grades
   private int gradeCounter; // number of grades entered
   private int aCount; // count of A grades
   private int bCount; // count of B grades
   private int cCount; // count of C grades
   private int dCount; // count of D grades
   private int fCount; // count of F grades

   // constructor initializes courseName;
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 1 of 3.)

```java
        return courseName;
    } // end method getCourseName

    // display a welcome message to the GradeBook user
    public void displayMessage()
    {
        // getCourseName gets the name of the course
        System.out.printf( "Welcome to the grade book for\n%s!\n\n",
            getCourseName() );
    } // end method displayMessage

    // input arbitrary number of grades from user
    public void inputGrades()
    {
        Scanner input = new Scanner( System.in );

        int grade; // grade entered by user

        System.out.printf( "%s\n%s\n   %s\n   %s\n",
            "Enter the integer grades in the range 0-100.",
            "Type the end-of-file indicator to terminate input:",
            "On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter",
            "On Windows type <Ctrl> z then press Enter" );

        // loop until user enters the end-of-file indicator
        while ( input.hasNext() )
        {
            grade = input.nextInt(); // read grade
            total += grade; // add grade to total
            ++gradeCounter; // increment number of grades

            // call method to increment appropriate counter
            incrementLetterGradeCounter( grade );
        } // end while
    } // end method inputGrades

    // add 1 to appropriate counter for specified grade
    private void incrementLetterGradeCounter( int grade )
    {
        // determine which grade was entered
        switch ( grade / 10 )
        {
            case 9:  // grade was between 90
            case 10: // and 100, inclusive
                ++aCount; // increment aCount
                break; // necessary to exit switch

            case 8: // grade was between 80 and 89
                ++bCount; // increment bCount
                break; // exit switch

```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 2 of 3.)

```
83          case 7: // grade was between 70 and 79
84             ++cCount; // increment cCount
85             break; // exit switch
86
87          case 6: // grade was between 60 and 69
88             ++dCount; // increment dCount
89             break; // exit switch
90
91          default: // grade was less than 60
92             ++fCount; // increment fCount
93             break; // optional; will exit switch anyway
94       } // end switch
95    } // end method incrementLetterGradeCounter
96
97    // display a report based on the grades entered by the user
98    public void displayGradeReport()
99    {
100      System.out.println( "\nGrade Report:" );
101
102      // if user entered at least one grade...
103      if ( gradeCounter != 0 )
104      {
105         // calculate average of all grades entered
106         double average = (double) total / gradeCounter;
107
108         // output summary of results
109         System.out.printf( "Total of the %d grades entered is %d\n",
110            gradeCounter, total );
111         System.out.printf( "Class average is %.2f\n", average );
112         System.out.printf( "%s\n%s%d\n%s%d\n%s%d\n%s%d\n%s%d\n",
113            "Number of students who received each grade:",
114            "A: ", aCount,   // display number of A grades
115            "B: ", bCount,   // display number of B grades
116            "C: ", cCount,   // display number of C grades
117            "D: ", dCount,   // display number of D grades
118            "F: ", fCount ); // display number of F grades
119      } // end if
120      else // no grades were entered, so output appropriate message
121         System.out.println( "No grades were entered" );
122   } // end method displayGradeReport
123 } // end class GradeBook
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 3 of 3.)

Like earlier versions of the class, class `GradeBook` (Fig. 5.9) declares instance variable `courseName` (line 7) and contains methods `setCourseName` (lines 24–27), `getCourseName` (lines 30–33) and `displayMessage` (lines 36–41), which set the course name, store the course name and display a welcome message to the user, respectively. The class also contains a constructor (lines 18–21) that initializes the course name.

Class `GradeBook` also declares instance variables `total` (line 9) and `gradeCounter` (line 10), which keep track of the sum of the grades entered by the user and the number of grades entered, respectively. Lines 11–15 declare counter variables for each grade category. Class `GradeBook` maintains `total`, `gradeCounter` and the five letter-grade counters as instance variables so that they can be used or modified in any of the class's methods. The class's constructor (lines 18–21) sets only the course name, because the remaining seven instance variables are `int`s and are initialized to `0` by default.

Class `GradeBook` (Fig. 5.9) contains three additional methods—`inputGrades`, `incrementLetterGradeCounter` and `displayGradeReport`. Method `inputGrades` (lines 44–66) reads an arbitrary number of integer grades from the user using sentinel-controlled repetition and updates instance variables `total` and `gradeCounter`. This method calls method `incrementLetterGradeCounter` (lines 69–95) to update the appropriate letter-grade counter for each grade entered. Method `displayGradeReport` (lines 98–122) outputs a report containing the total of all grades entered, the average of the grades and the number of students who received each letter grade. Let's examine these methods in more detail.

Method `inputGrades`

Line 48 in method `inputGrades` declares variable `grade`, which will store the user's input. Lines 50–54 prompt the user to enter integer grades and to type the end-of-file indicator to terminate the input. The **end-of-file indicator** is a system-dependent keystroke combination which the user enters to indicate that there's no more data to input. In Chapter 17, Files, Streams and Object Serialization, we'll see how the end-of-file indicator is used when a program reads its input from a file.

On UNIX/Linux/Mac OS X systems, end-of-file is entered by typing the sequence

<Ctrl> d

on a line by itself. This notation means to simultaneously press both the *Ctrl* key and the *d* key. On Windows systems, end-of-file can be entered by typing

<Ctrl> z

[*Note:* On some systems, you must press *Enter* after typing the end-of-file key sequence. Also, Windows typically displays the characters `^Z` on the screen when the end-of-file indicator is typed, as shown in the output of Fig. 5.10.]

Portability Tip 5.1

The keystroke combinations for entering end-of-file are system dependent.

The `while` statement (lines 57–65) obtains the user input. The condition at line 57 calls `Scanner` method **`hasNext`** to determine whether there's more data to input. This method returns the `boolean` value `true` if there's more data; otherwise, it returns `false`. The returned value is then used as the value of the condition in the `while` statement. Method `hasNext` returns `false` once the user types the end-of-file indicator.

Line 59 inputs a grade value from the user. Line 60 adds `grade` to `total`. Line 61 increments `gradeCounter`. The class's `displayGradeReport` method uses these variables to compute the average of the grades. Line 64 calls the class's `incrementLetterGradeCounter` method (declared in lines 69–95) to increment the appropriate letter-grade counter based on the numeric grade entered.

Method `incrementLetterGradeCounter`

Method `incrementLetterGradeCounter` contains a `switch` statement (lines 72–94) that determines which counter to increment. We assume that the user enters a valid grade in the range 0–100. A grade in the range 90–100 represents A, 80–89 represents B, 70–79 represents C, 60–69 represents D and 0–59 represents F. The switch statement consists of a block that contains a sequence of **case labels** and an optional **default case**. These are used in this example to determine which counter to increment based on the grade.

When the flow of control reaches the switch, the program evaluates the expression in the parentheses (grade / 10) following keyword switch. This is the switch's **controlling expression**. The program compares this expression's value (which must evaluate to an integral value of type byte, char, short or int) with each case label. The controlling expression in line 72 performs integer division, which *truncates the fractional part* of the result. Thus, when we divide a value from 0 to 100 by 10, the result is always a value from 0 to 10. We use several of these values in our case labels. For example, if the user enters the integer 85, the controlling expression evaluates to 8. The switch compares 8 with each case label. If a match occurs (case 8: at line 79), the program executes that case's statements. For the integer 8, line 80 increments bCount, because a grade in the 80s is a B. The **break statement** (line 81) causes program control to proceed with the first statement after the switch—in this program, we reach the end of method incrementLetterGradeCounter's body, so the method terminates and control returns to line 65 in method inputGrades (the first line after the call to incrementLetterGradeCounter). Line 65 is the end of a while loop's body, so control flows to the while's condition (line 57) to determine whether the loop should continue executing.

The cases in our switch explicitly test for the values 10, 9, 8, 7 and 6. Note the cases at lines 74–75 that test for the values 9 and 10 (both of which represent the grade A). Listing cases consecutively in this manner with no statements between them enables the cases to perform the same set of statements—when the controlling expression evaluates to 9 or 10, the statements in lines 76–77 will execute. The switch statement does not provide a mechanism for testing ranges of values, so every value you need to test must be listed in a separate case label. Each case can have multiple statements. The switch statement differs from other control statements in that it does *not* require braces around multiple statements in a case.

Without break statements, each time a match occurs in the switch, the statements for that case and subsequent cases execute until a break statement or the end of the switch is encountered. This is often referred to as "falling through" to the statements in subsequent cases. (This feature is perfect for writing a concise program that displays the iterative song "The Twelve Days of Christmas" in Exercise 5.29.)

Common Programming Error 5.6

Forgetting a break statement when one is needed in a switch is a logic error.

If no match occurs between the controlling expression's value and a case label, the default case (lines 91–93) executes. We use the default case in this example to process all controlling-expression values that are less than 6—that is, all failing grades. If no match occurs and the switch does not contain a default case, program control simply continues with the first statement after the switch.

GradeBookTest Class That Demonstrates Class GradeBook

Class GradeBookTest (Fig. 5.10) creates a GradeBook object (lines 10–11). Line 13 invokes the object's displayMessage method to output a welcome message to the user. Line 14 invokes the object's inputGrades method to read a set of grades from the user and keep track of the sum of all the grades entered and the number of grades. Recall that method `inputGrades` also calls method `incrementLetterGradeCounter` to keep track of the number of students who received each letter grade. Line 15 invokes method `displayGradeReport` of class `GradeBook`, which outputs a report based on the grades entered (as in the input/output window in Fig. 5.10). Line 103 of class `GradeBook` (Fig. 5.9) determines whether the user entered at least one grade—this helps us avoid dividing by zero. If so, line 106 calculates the average of the grades. Lines 109–118 then output the total of all the grades, the class average and the number of students who received each letter grade. If no grades were entered, line 121 outputs an appropriate message. The output in Fig. 5.10 shows a sample grade report based on 10 grades.

```
1  // Fig. 5.10: GradeBookTest.java
2  // Create GradeBook object, input grades and display grade report.
3
4  public class GradeBookTest
5  {
6     public static void main( String[] args )
7     {
8        // create GradeBook object myGradeBook and
9        // pass course name to constructor
10       GradeBook myGradeBook = new GradeBook(
11          "CS101 Introduction to Java Programming" );
12
13       myGradeBook.displayMessage(); // display welcome message
14       myGradeBook.inputGrades(); // read grades from user
15       myGradeBook.displayGradeReport(); // display report based on grades
16    } // end main
17 } // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter
   On Windows type <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80
```

Fig. 5.10 | Create `GradeBook` object, input grades and display grade report. (Part 1 of 2.)

```
Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Fig. 5.10 | Create `GradeBook` object, input grades and display grade report. (Part 2 of 2.)

Class `GradeBookTest` (Fig. 5.10) does not directly call `GradeBook` method `incrementLetterGradeCounter` (lines 69–95 of Fig. 5.9). This method is used exclusively by method `inputGrades` of class `GradeBook` to update the appropriate letter-grade counter as each new grade is entered by the user. Method `incrementLetterGradeCounter` exists solely to support the operations of `GradeBook`'s other methods, so it's declared `private`.

Software Engineering Observation 5.2

*Recall from Chapter 3 that methods declared with access modifier `private` can be called only by other methods of the class in which the `private` methods are declared. Such methods are commonly referred to as **utility methods** or **helper methods** because they're typically used to support the operation of the class's other methods.*

`switch` Statement UML Activity Diagram

Figure 5.11 shows the UML activity diagram for the general `switch` statement. Most `switch` statements use a `break` in each `case` to terminate the `switch` statement after processing the `case`. Figure 5.11 emphasizes this by including `break` statements in the activity

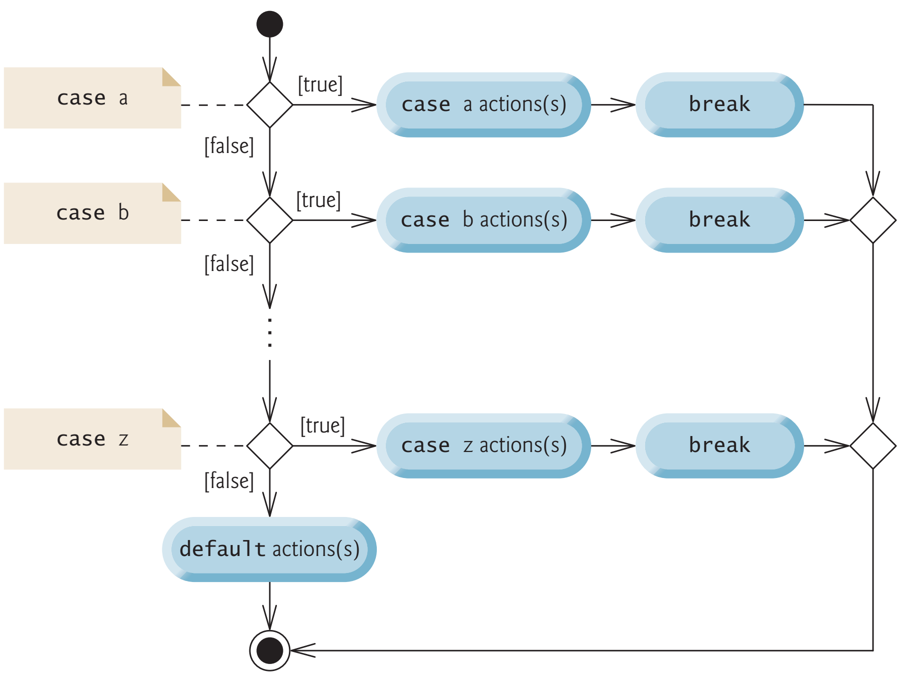


Fig. 5.11 | `switch` multiple-selection statement UML activity diagram with `break` statements.

diagram. The diagram makes it clear that the `break` statement at the end of a `case` causes control to exit the `switch` statement immediately.

The `break` statement is not required for the `switch`'s last `case` (or the optional `default` case, when it appears last), because execution continues with the next statement after the `switch`.

Software Engineering Observation 5.3

Provide a `default` case in `switch` statements. Including a `default` case focuses you on the need to process exceptional conditions.

Good Programming Practice 5.3

Although each `case` and the `default` case in a `switch` can occur in any order, place the `default` case last. When the `default` case is listed last, the `break` for that case is not required.

Notes on the Expression in Each `case` of a `switch`

When using the `switch` statement, remember that each `case` must contain a constant integral expression—that is, any combination of integer constants that evaluates to a constant integer value (e.g., –7, 0 or 221). An integer constant is simply an integer value. In addition, you can use **character constants**—specific characters in single quotes, such as `'A'`, `'7'` or `'$'`—which represent the integer values of characters and `enum` constants (introduced in Section 6.10). (Appendix B shows the integer values of the characters in the ASCII character set, which is a subset of the Unicode character set used by Java.)

The expression in each `case` can also be a **constant variable**—a variable containing a value which does not change for the entire program. Such a variable is declared with keyword `final` (discussed in Chapter 6). Java has a feature called *enumerations*, which we also present in Chapter 6. Enumeration constants can also be used in `case` labels. In Chapter 10, Object-Oriented Programming: Polymorphism, we present a more elegant way to implement `switch` logic—we use a technique called *polymorphism* to create programs that are often clearer, easier to maintain and easier to extend than programs using `switch` logic.

Using `String`s in `switch` Statements (New in Java SE 7)

As of Java SE 7, you can use `String`s in a `switch` statement's controlling expression and in `case` labels. For example, you might want to use a city's name to obtain the corresponding ZIP code. Assuming that `city` and `zipCode` are `String` variables, the following `switch` statement performs this task for three cities:

```
switch( city )
{
   case "Maynard":
      zipCode = "01754";
      break;
   case "Marlborough":
      zipCode = "01752";
      break;
   case "Framingham":
      zipCode = "01701";
      break;
} // end switch
```

5.7 break and continue Statements

In addition to selection and repetition statements, Java provides statements `break` and **continue** (presented in this section and Appendix O) to alter the flow of control. The preceding section showed how `break` can be used to terminate a `switch` statement's execution. This section discusses how to use `break` in repetition statements.

`break` Statement

The `break` statement, when executed in a `while`, `for`, `do…while` or `switch`, causes immediate exit from that statement. Execution continues with the first statement after the control statement. Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch` (as in Fig. 5.9). Figure 5.12 demonstrates a `break` statement exiting a `for`.

```
1  // Fig. 5.12: BreakTest.java
2  // break statement exiting a for statement.
3  public class BreakTest
4  {
5     public static void main( String[] args )
6     {
7        int count; // control variable also used after loop terminates
8
9        for ( count = 1; count <= 10; count++ ) // loop 10 times
10       {
11          if ( count == 5 ) // if count is 5,
12             break;         // terminate loop
13
14          System.out.printf( "%d ", count );
15       } // end for
16
17       System.out.printf( "\nBroke out of loop at count = %d\n", count );
18    } // end main
19 } // end class BreakTest
```

```
1 2 3 4
Broke out of loop at count = 5
```

Fig. 5.12 | `break` statement exiting a `for` statement.

When the `if` statement nested at lines 11–12 in the `for` statement (lines 9–15) detects that `count` is `5`, the `break` statement at line 12 executes. This terminates the `for` statement, and the program proceeds to line 17 (immediately after the `for` statement), which displays a message indicating the value of the control variable when the loop terminated. The loop fully executes its body only four times instead of 10.

`continue` Statement

The `continue` statement, when executed in a `while`, `for` or `do…while`, skips the remaining statements in the loop body and proceeds with the *next iteration* of the loop. In `while` and `do…while` statements, the program evaluates the loop-continuation test immediately after the `continue` statement executes. In a `for` statement, the increment expression executes, then the program evaluates the loop-continuation test.

```
1  // Fig. 5.13: ContinueTest.java
2  // continue statement terminating an iteration of a for statement.
3  public class ContinueTest
4  {
5     public static void main( String[] args )
6     {
7        for ( int count = 1; count <= 10; count++ ) // loop 10 times
8        {
9           if ( count == 5 ) // if count is 5,
10             continue; // skip remaining code in loop
11
12          System.out.printf( "%d ", count );
13       } // end for
14
15       System.out.println( "\nUsed continue to skip printing 5" );
16    } // end main
17 } // end class ContinueTest
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Fig. 5.13 | `continue` statement terminating an iteration of a `for` statement.

Figure 5.13 uses `continue` to skip the statement at line 12 when the nested `if` (line 9) determines that `count`'s value is `5`. When the `continue` statement executes, program control continues with the increment of the control variable in the `for` statement (line 7).

In Section 5.3, we stated that `while` could be used in most cases in place of `for`. This is *not* true when the increment expression in the `while` follows a `continue` statement. In this case, the increment does *not* execute before the program evaluates the repetition-continuation condition, so the `while` does not execute in the same manner as the `for`.

Software Engineering Observation 5.4

Some programmers feel that `break` and `continue` violate structured programming. Since the same effects are achievable with structured programming techniques, these programmers do not use `break` or `continue`.

Software Engineering Observation 5.5

There's a tension between achieving quality software engineering and achieving the best-performing software. Sometimes one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following rule of thumb: First, make your code simple and correct; then make it fast and small, but only if necessary.

5.8 Logical Operators

The `if`, `if…else`, `while`, `do…while` and `for` statements each require a condition to determine how to continue a program's flow of control. So far, we've studied only simple conditions, such as `count <= 10`, `number != sentinelValue` and `total > 1000`. Simple conditions are expressed in terms of the relational operators `>`, `<`, `>=` and `<=` and the equality operators `==` and `!=`, and each expression tests only one condition. To test multiple conditions in the process of making a decision, we performed these tests in separate statements or in nested `if` or `if...else` statements. Sometimes control statements require more complex conditions to determine a program's flow of control.

Java's **logical operators** enable you to form more complex conditions by *combining* simple conditions. The logical operators are `&&` (conditional AND), `||` (conditional OR), `&` (boolean logical AND), `|` (boolean logical inclusive OR), `^` (boolean logical exclusive OR) and `!` (logical NOT). [*Note:* The `&`, `|` and `^` operators are also bitwise operators when they're applied to integral operands. We discuss the bitwise operators in Appendix N.]

Conditional AND (&&) Operator

Suppose we wish to ensure at some point in a program that two conditions are *both* true before we choose a certain path of execution. In this case, we can use the **&&** (**conditional AND**) operator, as follows:

```
if ( gender == FEMALE && age >= 65 )
   ++seniorFemales;
```

This `if` statement contains two simple conditions. The condition `gender == FEMALE` compares variable `gender` to the constant `FEMALE` to determine whether a person is female. The condition `age >= 65` might be evaluated to determine whether a person is a senior citizen. The `if` statement considers the combined condition

```
gender == FEMALE && age >= 65
```

which is true if and only if *both* simple conditions are true. In this case, the `if` statement's body increments `seniorFemales` by `1`. If either or both of the simple conditions are false, the program skips the increment. Some programmers find that the preceding combined condition is more readable when redundant parentheses are added, as in:

```
( gender == FEMALE ) && ( age >= 65 )
```

The table in Fig. 5.14 summarizes the `&&` operator. The table shows all four possible combinations of `false` and `true` values for *expression1* and *expression2*. Such tables are called **truth tables**. Java evaluates to `false` or `true` all expressions that include relational operators, equality operators or logical operators.

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 5.14 | `&&` (conditional AND) operator truth table.

Conditional OR (||) Operator

Now suppose we wish to ensure that *either or both* of two conditions are true before we choose a certain path of execution. In this case, we use the **||** (**conditional OR**) operator, as in the following program segment:

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
   System.out.println ( "Student grade is A" );
```

This statement also contains two simple conditions. The condition `semesterAverage >= 90` evaluates to determine whether the student deserves an A in the course because of a solid performance throughout the semester. The condition `finalExam >= 90` evaluates to determine whether the student deserves an A in the course because of an outstanding performance on the final exam. The `if` statement then considers the combined condition

```
( semesterAverage >= 90 ) || ( finalExam >= 90 )
```

and awards the student an A if *either or both* of the simple conditions are true. The only time the message `"Student grade is A"` is *not* printed is when *both* of the simple conditions are *false*. Figure 5.15 is a truth table for operator conditional OR (`||`). Operator `&&` has a higher precedence than operator `||`. Both operators associate from left to right.

expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 5.15 | `||` (conditional OR) operator truth table.

Short-Circuit Evaluation of Complex Conditions

The parts of an expression containing `&&` or `||` operators are evaluated *only* until it's known whether the condition is true or false. Thus, evaluation of the expression

```
( gender == FEMALE ) && ( age >= 65 )
```

stops immediately if `gender` is not equal to `FEMALE` (i.e., the entire expression is `false`) and continues if `gender` *is* equal to `FEMALE` (i.e., the entire expression could still be `true` if the condition `age >= 65` is `true`). This feature of conditional AND and conditional OR expressions is called **short-circuit evaluation**.

Common Programming Error 5.7

In expressions using operator `&&`, a condition—we'll call this the dependent condition—may require another condition to be true for the evaluation of the dependent condition to be meaningful. In this case, the dependent condition should be placed after *the other condition, or an error might occur. For example, in the expression `( i != 0 ) && ( 10 / i == 2 )`, the second condition must appear after the first condition, or a divide-by-zero error might occur.*

Boolean Logical AND (&) and Boolean Logical Inclusive OR (|) Operators

The **boolean logical AND** (`&`) and **boolean logical inclusive OR** (`|`) operators are identical to the `&&` and `||` operators, except that the `&` and `|` operators *always* evaluate *both* of their operands (i.e., they do *not* perform short-circuit evaluation). So, the expression

```
( gender == 1 ) & ( age >= 65 )
```

evaluates `age >= 65` *regardless* of whether `gender` is equal to `1`. This is useful if the right operand of the boolean logical AND or boolean logical inclusive OR operator has a required **side effect**—a modification of a variable's value. For example, the expression

```
    ( birthday == true ) | ( ++age >= 65 )
```

guarantees that the condition `++age >= 65` will be evaluated. Thus, the variable `age` is incremented, regardless of whether the overall expression is `true` or `false`.

Error-Prevention Tip 5.6

For clarity, avoid expressions with side effects in conditions. The side effects may seem clever, but they can make it harder to understand code and can lead to subtle logic errors.

Boolean Logical Exclusive OR (^)

A simple condition containing the **boolean logical exclusive OR** (`^`) operator is `true` *if and only if one of its operands is* `true` *and the other is* `false`. If both are `true` or both are `false`, the entire condition is `false`. Figure 5.16 is a truth table for the boolean logical exclusive OR operator (`^`). This operator is guaranteed to evaluate *both* of its operands.

expression1	expression2	expression1 ^ expression2
false	false	false
false	true	true
true	false	true
true	true	false

Fig. 5.16 | `^` (boolean logical exclusive OR) operator truth table.

Logical Negation (!) Operator

The `!` (**logical NOT**, also called **logical negation** or **logical complement**) operator "reverses" the meaning of a condition. Unlike the logical operators `&&`, `||`, `&`, `|` and `^`, which are ***binary*** operators that combine two conditions, the logical negation operator is a *unary* operator that has only a single condition as an operand. The operator is placed *before* a condition to choose a path of execution if the original condition (without the logical negation operator) is `false`, as in the program segment

```
    if ( ! ( grade == sentinelValue ) )
       System.out.printf( "The next grade is %d\n", grade );
```

which executes the `printf` call only if `grade` is *not* equal to `sentinelValue`. The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a higher precedence than the equality operator.

In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the previous statement may also be written as follows:

```
    if ( grade != sentinelValue )
       System.out.printf( "The next grade is %d\n", grade );
```

This flexibility can help you express a condition in a more convenient manner. Figure 5.17 is a truth table for the logical negation operator.

expression	!expression
`false`	`true`
`true`	`false`

Fig. 5.17 | `!` (logical negation, or logical NOT) operator truth table.

Logical Operators Example

Figure 5.18 uses logical operators to produce the truth tables discussed in this section. The output shows the `boolean` expression that was evaluated and its result. We used the **`%b` format specifier** to display the word “true” or the word “false” based on a `boolean` expression’s value. Lines 9–13 produce the truth table for `&&`. Lines 16–20 produce the truth table for `||`. Lines 23–27 produce the truth table for `&`. Lines 30–35 produce the truth table for `|`. Lines 38–43 produce the truth table for `^`. Lines 46–47 produce the truth table for `!`.

```
1  // Fig. 5.18: LogicalOperators.java
2  // Logical operators.
3
4  public class LogicalOperators
5  {
6     public static void main( String[] args )
7     {
8        // create truth table for && (conditional AND) operator
9        System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
10          "Conditional AND (&&)", "false && false", ( false && false ),
11          "false && true", ( false && true ),
12          "true && false", ( true && false ),
13          "true && true", ( true && true ) );
14
15       // create truth table for || (conditional OR) operator
16       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
17          "Conditional OR (||)", "false || false", ( false || false ),
18          "false || true", ( false || true ),
19          "true || false", ( true || false ),
20          "true || true", ( true || true ) );
21
22       // create truth table for & (boolean logical AND) operator
23       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
24          "Boolean logical AND (&)", "false & false", ( false & false ),
25          "false & true", ( false & true ),
26          "true & false", ( true & false ),
27          "true & true", ( true & true ) );
28
29       // create truth table for | (boolean logical inclusive OR) operator
30       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
31          "Boolean logical inclusive OR (|)",
```

Fig. 5.18 | Logical operators. (Part 1 of 2.)

```java
            "false | false", ( false | false ),
            "false | true", ( false | true ),
            "true | false", ( true | false ),
            "true | true", ( true | true ) );

         // create truth table for ^ (boolean logical exclusive OR) operator
         System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
            "Boolean logical exclusive OR (^)",
            "false ^ false", ( false ^ false ),
            "false ^ true", ( false ^ true ),
            "true ^ false", ( true ^ false ),
            "true ^ true", ( true ^ true ) );

         // create truth table for ! (logical negation) operator
         System.out.printf( "%s\n%s: %b\n%s: %b\n", "Logical NOT (!)",
            "!false", ( !false ), "!true", ( !true ) );
      } // end main
   } // end class LogicalOperators
```

```
Conditional AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Conditional OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Boolean logical AND (&)
false & false: false
false & true: false
true & false: false
true & true: true

Boolean logical inclusive OR (|)
false | false: false
false | true: true
true | false: true
true | true: true

Boolean logical exclusive OR (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

Logical NOT (!)
!false: true
!true: false
```

Fig. 5.18 | Logical operators. (Part 2 of 2.)

Figure 5.19 shows the precedence and associativity of the Java operators introduced so far. The operators are shown from top to bottom in decreasing order of precedence.

Operators	Associativity	Type
`++ --`	right to left	unary postfix
`++ -- + - !` (*type*)	right to left	unary prefix
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`&`	left to right	boolean logical AND
`^`	left to right	boolean logical exclusive OR
`|`	left to right	boolean logical inclusive OR
`&&`	left to right	conditional AND
`||`	left to right	conditional OR
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment

Fig. 5.19 | Precedence/associativity of the operators discussed so far.

5.9 Structured Programming Summary

Just as architects design buildings by employing the collective wisdom of their profession, so should programmers design programs. Our field is much younger than architecture, and our collective wisdom is considerably sparser. We've learned that structured programming produces programs that are easier than unstructured programs to understand, test, debug, modify and even prove correct in a mathematical sense.

Figure 5.20 uses UML activity diagrams to summarize Java's control statements. The initial and final states indicate the *single entry point* and the *single exit point* of each control statement. Arbitrarily connecting individual symbols in an activity diagram can lead to unstructured programs. Therefore, the programming profession has chosen a limited set of control statements that can be combined in only two simple ways to build structured programs.

For simplicity, Java includes only *single-entry/single-exit* control statements—there's only one way to enter and only one way to exit each control statement. Connecting control statements in sequence to form structured programs is simple. The final state of one control statement is connected to the initial state of the next—that is, the control statements are placed one after another in a program in sequence. We call this *control-statement stacking*. The rules for forming structured programs also allow for control statements to be *nested*.

Figure 5.21 shows the rules for forming structured programs. The rules assume that action states may be used to indicate any action. The rules also assume that we begin with the simplest activity diagram (Fig. 5.22) consisting of only an initial state, an action state, a final state and transition arrows.

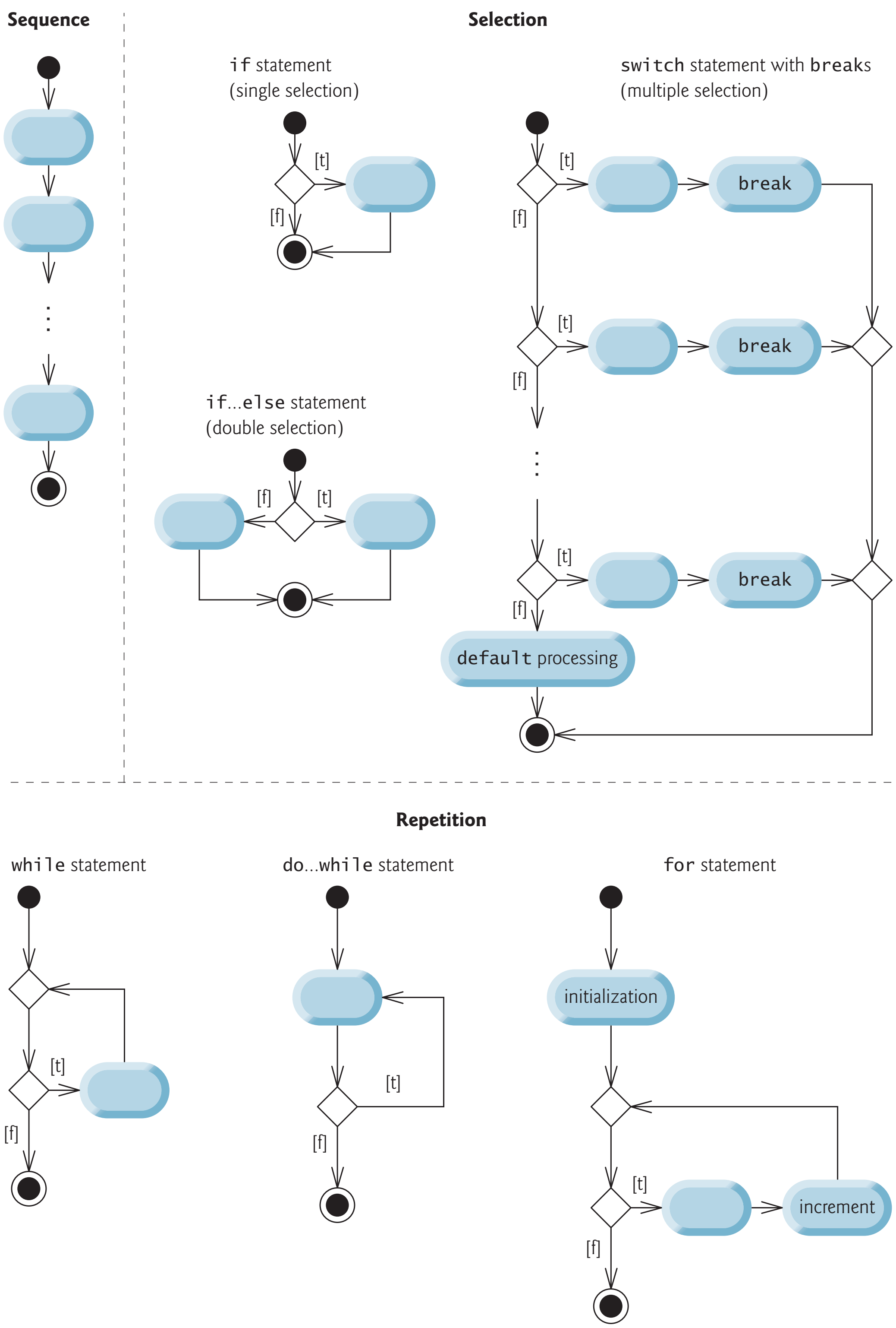


Fig. 5.20 | Java's single-entry/single-exit sequence, selection and repetition statements.

Rules for forming structured programs

1. Begin with the simplest activity diagram (Fig. 5.22).
2. Any action state can be replaced by two action states in sequence.
3. Any action state can be replaced by any control statement (sequence of action states, `if`, `if...else`, `switch`, `while`, `do...while` or `for`).
4. Rules 2 and 3 can be applied as often as you like and in any order.

Fig. 5.21 | Rules for forming structured programs.

Applying the rules in Fig. 5.21 always results in a properly structured activity diagram with a neat, building-block appearance. For example, repeatedly applying rule 2 to the simplest activity diagram results in an activity diagram containing many action states in sequence (Fig. 5.23). Rule 2 generates a stack of control statements, so let's call rule 2 the **stacking rule**. The vertical dashed lines in Fig. 5.23 are not part of the UML—we use them to separate the four activity diagrams that demonstrate rule 2 of Fig. 5.21 being applied.

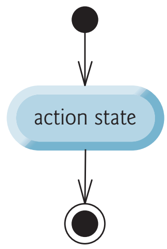


Fig. 5.22 | Simplest activity diagram.

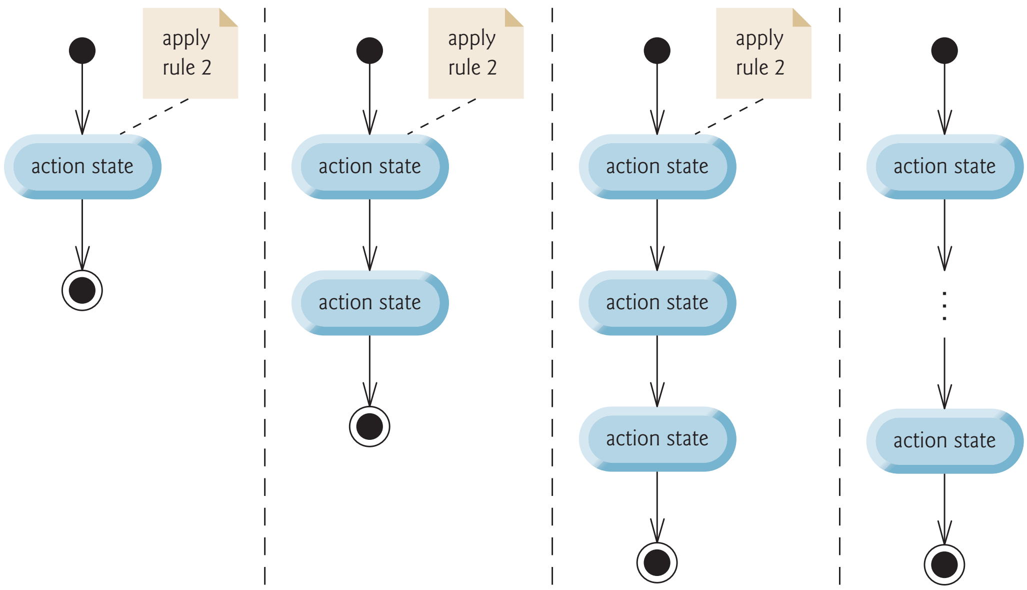


Fig. 5.23 | Repeatedly applying the Rule 2 of Fig. 5.21 to the simplest activity diagram.

Rule 3 is called the **nesting rule**. Repeatedly applying rule 3 to the simplest activity diagram results in one with neatly nested control statements. For example, in Fig. 5.24, the action state in the simplest activity diagram is replaced with a double-selection (`if...else`) statement. Then rule 3 is applied again to the action states in the double-selection statement, replacing each with a double-selection statement. The dashed action-state symbol around each double-selection statement represents the action state that was replaced. [*Note:* The dashed arrows and dashed action-state symbols shown in Fig. 5.24 are not part of the UML. They're used here to illustrate that any action state can be replaced with a control statement.]

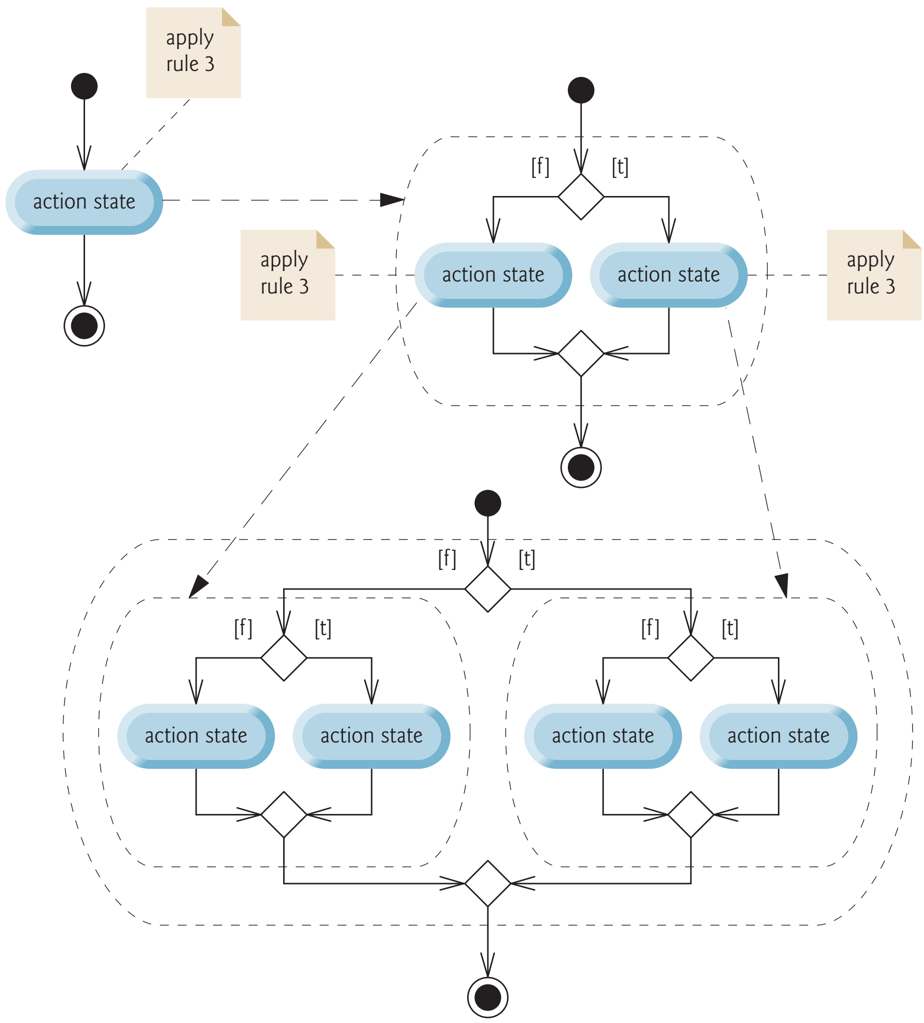


Fig. 5.24 | Repeatedly applying the Rule 3 of Fig. 5.21 to the simplest activity diagram.

Rule 4 generates larger, more involved and more deeply nested statements. The diagrams that emerge from applying the rules in Fig. 5.21 constitute the set of all possible structured activity diagrams and hence the set of all possible structured programs. The beauty of the structured approach is that we use *only seven* simple single-entry/single-exit control statements and assemble them in *only two* simple ways.

If the rules in Fig. 5.21 are followed, an "unstructured' activity diagram (like the one in Fig. 5.25) cannot be created. If you're uncertain about whether a particular diagram is structured, apply the rules of Fig. 5.21 in reverse to reduce it to the simplest activity diagram. If you can reduce it, the original diagram is structured; otherwise, it's not.

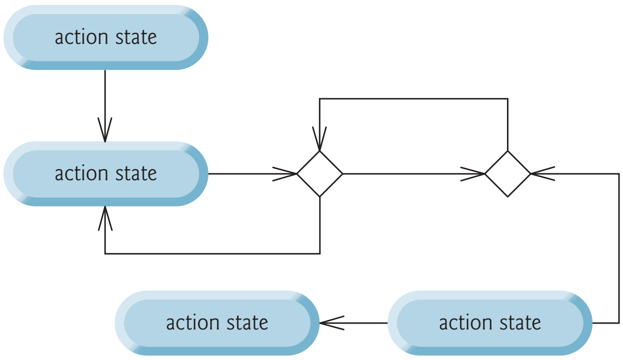


Fig. 5.25 | "Unstructured" activity diagram.

Structured programming promotes simplicity. Only three forms of control are needed to implement an algorithm:

- Sequence
- Selection
- Repetition

The sequence structure is trivial. Simply list the statements to execute in the order in which they should execute. Selection is implemented in one of three ways:

- `if` statement (single selection)
- `if…else` statement (double selection)
- `switch` statement (multiple selection)

In fact, it's straightforward to prove that the simple `if` statement is sufficient to provide *any* form of selection—everything that can be done with the `if…else` statement and the `switch` statement can be implemented by combining `if` statements (although perhaps not as clearly and efficiently).

Repetition is implemented in one of three ways:

- `while` statement
- `do…while` statement
- `for` statement

[*Note:* There's a fourth repetition statement—the enhanced `for` statement—that we discuss in Section 7.6.] It's straightforward to prove that the `while` statement is sufficient to provide *any* form of repetition. Everything that can be done with `do…while` and `for` can be done with the `while` statement (although perhaps not as conveniently).

Combining these results illustrates that *any* form of control ever needed in a Java program can be expressed in terms of

- sequence
- `if` statement (selection)
- `while` statement (repetition)

and that these can be combined in only two ways—*stacking* and *nesting*. Indeed, structured programming is the essence of simplicity.

5.10 (Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals

This section demonstrates drawing rectangles and ovals, using the `Graphics` methods **`drawRect`** and **`drawOval`**, respectively. These methods are demonstrated in Fig. 5.26.

```java
// Fig. 5.26: Shapes.java
// Demonstrates drawing different shapes.
import java.awt.Graphics;
import javax.swing.JPanel;

public class Shapes extends JPanel
{
   private int choice; // user's choice of which shape to draw

   // constructor sets the user's choice
   public Shapes( int userChoice )
   {
      choice = userChoice;
   } // end Shapes constructor

   // draws a cascade of shapes starting from the top-left corner
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

      for ( int i = 0; i < 10; i++ )
      {
         // pick the shape based on the user's choice
         switch ( choice )
         {
            case 1: // draw rectangles
               g.drawRect( 10 + i * 10, 10 + i * 10,
                  50 + i * 10, 50 + i * 10 );
               break;
            case 2: // draw ovals
               g.drawOval( 10 + i * 10, 10 + i * 10,
                  50 + i * 10, 50 + i * 10 );
               break;
         } // end switch
      } // end for
   } // end method paintComponent
} // end class Shapes
```

Fig. 5.26 | Drawing a cascade of shapes based on the user's choice.

Line 6 begins the class declaration for `Shapes`, which extends `JPanel`. Instance variable `choice`, declared in line 8, determines whether `paintComponent` should draw rectangles or ovals. The `Shapes` constructor at lines 11–14 initializes `choice` with the value passed in parameter `userChoice`.

Method `paintComponent` (lines 17–36) performs the actual drawing. Remember, the first statement in every `paintComponent` method should be a call to `super.paintComponent`, as in line 19. Lines 21–35 loop 10 times to draw 10 shapes. The nested `switch` statement (lines 24–34) chooses between drawing rectangles and drawing ovals.

If `choice` is 1, then the program draws rectangles. Lines 27–28 call `Graphics` method `drawRect`. Method `drawRect` requires four arguments. The first two represent the *x*- and *y*-coordinates of the upper-left corner of the rectangle; the next two represent the rectangle's width and height. In this example, we start at a position 10 pixels down and 10 pixels right of the top-left corner, and every iteration of the loop moves the upper-left corner another 10 pixels down and to the right. The width and the height of the rectangle start at 50 pixels and increase by 10 pixels in each iteration.

If `choice` is 2, the program draws ovals. It creates an imaginary rectangle called a **bounding rectangle** and places inside it an oval that touches the midpoints of all four sides. Method `drawOval` (lines 31–32) requires the same four arguments as method `drawRect`. The arguments specify the position and size of the bounding rectangle for the oval. The values passed to `drawOval` in this example are exactly the same as those passed to `drawRect` in lines 27–28. Since the width and height of the bounding rectangle are identical in this example, lines 27–28 draw a circle. As an exercise, try modifying the program to draw both rectangles and ovals to see how `drawOval` and `drawRect` are related.

Class ShapesTest

Figure 5.27 is responsible for handling input from the user and creating a window to display the appropriate drawing based on the user's response. Line 3 imports `JFrame` to handle the display, and line 4 imports `JOptionPane` to handle the input.

```
1  // Fig. 5.27: ShapesTest.java
2  // Test application that displays class Shapes.
3  import javax.swing.JFrame;
4  import javax.swing.JOptionPane;
5
6  public class ShapesTest
7  {
8     public static void main( String[] args )
9     {
10       // obtain user's choice
11       String input = JOptionPane.showInputDialog(
12          "Enter 1 to draw rectangles\n" +
13          "Enter 2 to draw ovals" );
14
15       int choice = Integer.parseInt( input ); // convert input to int
16
17       // create the panel with the user's input
18       Shapes panel = new Shapes( choice );
```

Fig. 5.27 | Obtaining user input and creating a `JFrame` to display `Shapes`. (Part 1 of 2.)

```
19
20         JFrame application = new JFrame(); // creates a new JFrame
21
22         application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
23         application.add( panel ); // add the panel to the frame
24         application.setSize( 300, 300 ); // set the desired size
25         application.setVisible( true ); // show the frame
26      } // end main
27   } // end class ShapesTest
```

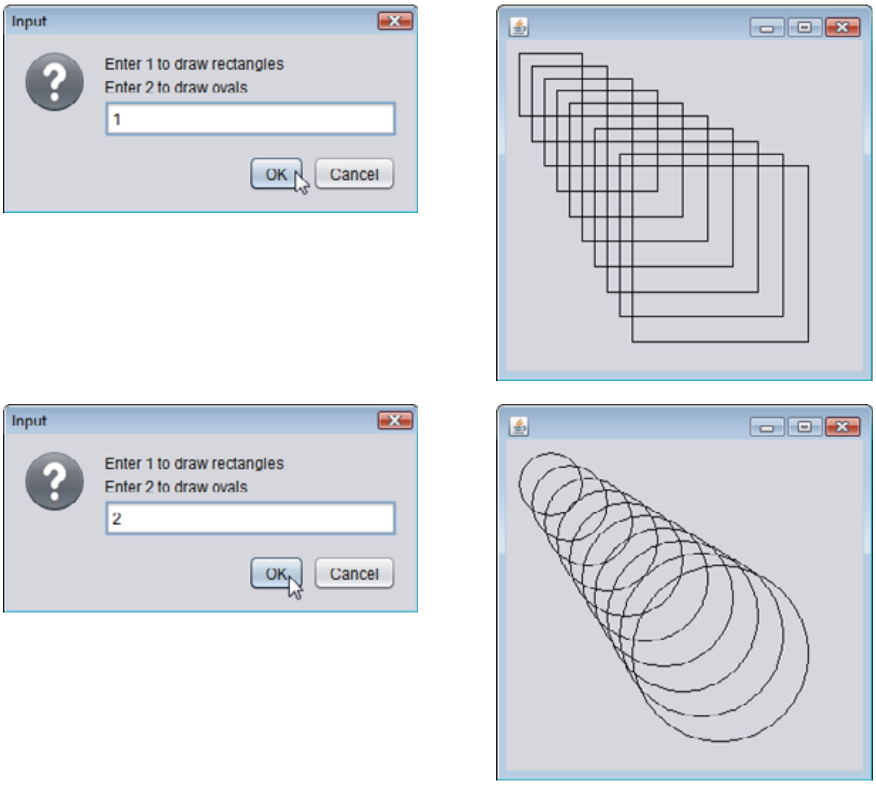


Fig. 5.27 | Obtaining user input and creating a `JFrame` to display `Shapes`. (Part 2 of 2.)

Lines 11–13 prompt the user with an input dialog and store the user's response in variable `input`. Line 15 uses `Integer` method `parseInt` to convert the `String` entered by the user to an `int` and stores the result in variable `choice`. Line 18 creates a `Shapes` object and passes the user's choice to the constructor. Lines 20–25 perform the standard operations that create and set up a window in this case study—create a frame, set it to exit the application when closed, add the drawing to the frame, set the frame size and make it visible.

GUI and Graphics Case Study Exercises

5.1 Draw 12 concentric circles in the center of a `JPanel` (Fig. 5.28). The innermost circle should have a radius of 10 pixels, and each successive circle should have a radius 10 pixels larger than the previous one. Begin by finding the center of the `JPanel`. To get the upper-left corner of a circle, move up one radius and to the left one radius from the center. The width and height of the bounding rectangle are both the same as the circle's diameter (i.e., twice the radius).

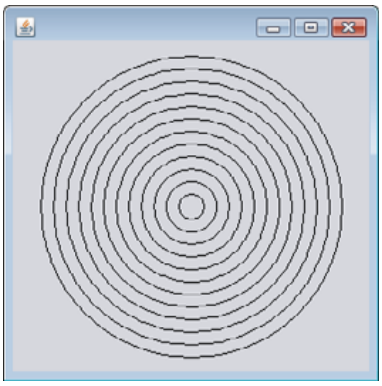


Fig. 5.28 | Drawing concentric circles.

5.2 Modify Exercise 5.16 from the end-of-chapter exercises to read input using dialogs and to display the bar chart using rectangles of varying lengths.

5.11 Wrap-Up

In this chapter, we completed our introduction to Java's control statements, which enable you to control the flow of execution in methods. Chapter 4 discussed Java's `if`, `if…else` and `while` statements. The current chapter demonstrated the `for`, `do…while` and `switch` statements. We showed that any algorithm can be developed using combinations of the sequence structure (i.e., statements listed in the order in which they should execute), the three types of selection statements—`if`, `if…else` and `switch`—and the three types of repetition statements—`while`, `do…while` and `for`. In this chapter and Chapter 4, we discussed how you can combine these building blocks to utilize proven program-construction and problem-solving techniques. This chapter also introduced Java's logical operators, which enable you to use more complex conditional expressions in control statements. In Chapter 6, we examine methods in greater depth.

Summary

Section 5.2 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition (p. 152) requires a control variable, the initial value of the control variable, the increment (or decrement) by which the control variable is modified each time through the loop (also known as each iteration of the loop) and the loop-continuation condition that determines whether looping should continue.
- You can declare a variable and initialize it in the same statement.

Section 5.3 `for` Repetition Statement

- The `while` statement can be used to implement any counter-controlled loop.
- The `for` statement (p. 154) specifies all the details of counter-controlled repetition in its header
- When the `for` statement begins executing, its control variable is declared and initialized. Next, the program checks the loop-continuation condition. If the condition is initially true, the body executes. After executing the loop's body, the increment expression executes. Then the loop-con-

tinuation test is performed again to determine whether the program should continue with the next iteration of the loop.

- The general format of the `for` statement is

  `for (` *initialization*`;` *loopContinuationCondition*`;` *increment* `)`
      statement

  where the *initialization* expression names the loop's control variable and provides its initial value, *loopContinuationCondition* determines whether the loop should continue executing and *increment* modifies the control variable's value, so that the loop-continuation condition eventually becomes false. The two semicolons in the `for` header are required.
- Most `for` statements can be represented with equivalent `while` statements as follows:

  initialization;

  `while (` *loopContinuationCondition* `)`
  `{`
      statement
      increment;
  `}`
- Typically, `for` statements are used for counter-controlled repetition and `while` statements for sentinel-controlled repetition.
- If the *initialization* expression in the `for` header declares the control variable, the control variable can be used only in that `for` statement—it will not exist outside the `for` statement.
- The expressions in a `for` header are optional. If the *loopContinuationCondition* is omitted, Java assumes that it's always true, thus creating an infinite loop. You might omit the *initialization* expression if the control variable is initialized before the loop. You might omit the *increment* expression if the increment is calculated with statements in the loop's body or if no increment is needed.
- The increment expression in a `for` acts as if it's a standalone statement at the end of the `for`'s body.
- A `for` statement can count downward by using a negative increment (i.e., a decrement).
- If the loop-continuation condition is initially `false`, the program does not execute the `for` statement's body. Instead, execution proceeds with the statement following the `for`.

Section 5.4 Examples Using the `for` *Statement*

- Java treats floating-point constants like `1000.0` and `0.05` as type `double`. Similarly, Java treats whole-number constants like `7` and `-22` as type `int`.
- The format specifier `%4s` outputs a `String` in a field width (p. 160) of 4—that is, `printf` displays the value with at least 4 character positions. If the value to be output is less than 4 character positions wide, the value is right justified (p. 160) in the field by default. If the value is greater than 4 character positions wide, the field width expands to accommodate the appropriate number of characters. To left justify (p. 160) the value, use a negative integer to specify the field width.
- `Math.pow(`x`,` y`)` (p. 161) calculates the value of x raised to the y^{th} power. The method receives two `double` arguments and returns a `double` value.
- The comma (`,`) formatting flag (p. 161) in a format specifier indicates that a floating-point value should be output with a grouping separator (p. 161). The actual separator used is specific to the user's locale (i.e., country). In the United States, the number will have commas separating every three digits and a decimal point separating the fractional part of the number, as in 1,234.45.
- The `.` in a format specifier indicates that the integer to its right is the number's precision.

Section 5.5 `do…while` *Repetition Statement*

- The `do…while` statement (p. 162) is similar to the `while` statement. In the `while`, the program tests the loop-continuation condition at the beginning of the loop, before executing its body; if the condition is false, the body never executes. The `do…while` statement tests the loop-continuation condition *after* executing the loop's body; therefore, the body always executes at least once.

Section 5.6 `switch` Multiple-Selection Statement

- The `switch` statement (p. 164) performs different actions based on the possible values of a constant integral expression (a constant value of type `byte`, `short`, `int` or `char`, but not `long`).
- The end-of-file indicator (p. 167) is a system-dependent keystroke combination that terminates user input. On UNIX/Linux/Mac OS X systems, end-of-file is entered by typing the sequence <*Ctrl*> *d* on a line by itself. This notation means to simultaneously press both the *Ctrl* key and the *d* key. On Windows systems, enter end-of-file by typing <*Ctrl*> *z*.
- `Scanner` method `hasNext` (p. 167) determines whether there's more data to input. This method returns the `boolean` value `true` if there's more data; otherwise, it returns `false`. As long as the end-of-file indicator has not been typed, method `hasNext` will return `true`.
- The `switch` statement consists of a block that contains a sequence of `case` labels (p. 168) and an optional `default` case (p. 168).
- When the flow of control reaches a `switch`, the program evaluates the `switch`'s controlling expression and compares its value with each `case` label. If a match occurs, the program executes the statements for that `case`.
- Listing cases consecutively with no statements between them enables the cases to perform the same set of statements.
- Every value you wish to test in a `switch` must be listed in a separate `case` label.
- Each `case` can have multiple statements, and these need not be placed in braces.
- Without `break` statements, each time a match occurs in the `switch`, the statements for that case and subsequent cases execute until a `break` statement or the end of the `switch` is encountered.
- If no match occurs between the controlling expression's value and a `case` label, the optional `default` case executes. If no match occurs and the `switch` does not contain a `default` case, program control simply continues with the first statement after the `switch`.
- As of Java SE 7, you can use `String`s in a `switch` statement's controlling expression and `case` labels.

Section 5.7 `break` and `continue` Statements

- The `break` statement (p. 168), when executed in a `while`, `for`, `do…while` or `switch`, causes immediate exit from that statement.
- The `continue` statement (p. 172), when executed in a `while`, `for` or `do…while`, skips the loop's remaining body statements and proceeds with its next iteration. In `while` and `do…while` statements, the program evaluates the loop-continuation test immediately. In a `for` statement, the increment expression executes, then the program evaluates the loop-continuation test.

Section 5.8 Logical Operators

- Simple conditions are expressed in terms of the relational operators `>`, `<`, `>=` and `<=` and the equality operators `==` and `!=`, and each expression tests only one condition.
- Logical operators (p. 174) enable you to form more complex conditions by combining simple conditions. The logical operators are `&&` (conditional AND), `||` (conditional OR), `&` (boolean logical AND), `|` (boolean logical inclusive OR), `^` (boolean logical exclusive OR) and `!` (logical NOT).
- To ensure that two conditions are *both* true, use the `&&` (conditional AND) operator. If either or both of the simple conditions are false, the entire expression is false.
- To ensure that either *or* both of two conditions are true, use the `||` (conditional OR) operator, which evaluates to true if either or both of its simple conditions are true.

- A condition using `&&` or `||` operators (p. 174) uses short-circuit evaluation (p. 175)—they're evaluated only until it's known whether the condition is true or false.
- The `&` and `|` operators (p. 175) work identically to the `&&` and `||` operators, but always evaluate both operands.
- A simple condition containing the boolean logical exclusive OR (`^`; p. 176) operator is `true` *if and only if one of its operands is* `true` *and the other is* `false`. If both operands are `true` or both are `false`, the entire condition is `false`. This operator is also guaranteed to evaluate both of its operands.
- The unary `!` (logical NOT; p. 176) operator "reverses" the value of a condition.

Self-Review Exercises

5.1 *(Fill in the Blanks)* Fill in the blanks in each of the following statements:

a) Typically, ________ statements are used for counter-controlled repetition and ________ statements for sentinel-controlled repetition.
b) The `do…while` statement tests the loop-continuation condition ________ executing the loop's body; therefore, the body always executes at least once.
c) The ________ statement selects among multiple actions based on the possible values of an integer variable or expression.
d) The ________ statement, when executed in a repetition statement, skips the remaining statements in the loop body and proceeds with the next iteration of the loop.
e) The ________ operator can be used to ensure that two conditions are *both* true before choosing a certain path of execution.
f) If the loop-continuation condition in a `for` header is initially ________, the program does not execute the `for` statement's body.
g) Methods that perform common tasks and do not require objects are called ________ methods.

5.2 *(True/False Questions)* State whether each of the following is *true* or *false*. If *false*, explain why.

a) The `default` case is required in the `switch` selection statement.
b) The `break` statement is required in the last case of a `switch` selection statement.
c) The expression `( ( x > y ) && ( a < b ) )` is true if either `x > y` is true or `a < b` is true.
d) An expression containing the `||` operator is true if either or both of its operands are true.
e) The comma (`,`) formatting flag in a format specifier (e.g., `%,20.2f`) indicates that a value should be output with a thousands separator.
f) To test for a range of values in a `switch` statement, use a hyphen (–) between the start and end values of the range in a `case` label.
g) Listing cases consecutively with no statements between them enables the cases to perform the same set of statements.

5.3 *(Write a Statement)* Write a Java statement or a set of Java statements to accomplish each of the following tasks:

a) Sum the odd integers between 1 and 99, using a `for` statement. Assume that the integer variables `sum` and `count` have been declared.
b) Calculate the value of `2.5` raised to the power of `3`, using the `pow` method.
c) Print the integers from 1 to 20, using a `while` loop and the counter variable `i`. Assume that the variable `i` has been declared, but not initialized. Print only five integers per line. [*Hint:* Use the calculation `i % 5`. When the value of this expression is 0, print a newline character; otherwise, print a tab character. Assume that this code is an application. Use the `System.out.println()` method to output the newline character, and use the `System.out.print( '\t' )` method to output the tab character.]
d) Repeat part (c), using a `for` statement.

5.4 *(Find the Error)* Find the error in each of the following code segments, and explain how to correct it:

a)
```
i = 1;

while ( i <= 10 );
   ++i;
}
```
b)
```
for ( k = 0.1; k != 1.0; k += 0.1 )
   System.out.println( k );
```
c)
```
switch ( n )
{
   case 1:
      System.out.println( "The number is 1" );
   case 2:
      System.out.println( "The number is 2" );
      break;
   default:
      System.out.println( "The number is not 1 or 2" );
      break;
}
```
d) The following code should print the values 1 to 10:
```
n = 1;
while ( n < 10 )
   System.out.println( n++ );
```

Answers to Self-Review Exercises

5.1 a) `for`, `while`. b) after. c) `switch`. d) `continue`. e) `&&` (conditional AND). f) `false`. g) `static`.

5.2 a) False. The `default` case is optional. If no default action is needed, then there's no need for a `default` case. b) False. The `break` statement is used to exit the `switch` statement. The `break` statement is not required for the last case in a `switch` statement. c) False. Both of the relational expressions must be true for the entire expression to be true when using the `&&` operator. d) True. e) True. f) False. The `switch` statement does not provide a mechanism for testing ranges of values, so every value that must be tested should be listed in a separate `case` label. g) True.

5.3 a)
```
sum = 0;
for ( count = 1; count <= 99; count += 2 )
   sum += count;
```
b)
```
double result = Math.pow( 2.5, 3 );
```
e)
```
i = 1;

while ( i <= 20 )
{
   System.out.print( i );

   if ( i % 5 == 0 )
      System.out.println();
   else
      System.out.print( '\t' );

   ++i;
}
```
f)
```
for ( i = 1; i <= 20; i++ )
{
    System.out.print( i );
```

```
    if ( i % 5 == 0 )
       System.out.println();
    else
       System.out.print( '\t' );
 }
```

5.4 a) Error: The semicolon after the `while` header causes an infinite loop, and there's a missing left brace.
Correction: Replace the semicolon by a `{`, or remove both the `;` and the `}`.

b) Error: Using a floating-point number to control a `for` statement may not work, because floating-point numbers are represented only approximately by most computers.
Correction: Use an integer, and perform the proper calculation in order to get the values you desire:

```
for ( k = 1; k != 10; k++ )
   System.out.println( (double) k / 10 );
```

c) Error: The missing code is the `break` statement in the statements for the first `case`.
Correction: Add a `break` statement at the end of the statements for the first `case`. This omission is not necessarily an error if you want the statement of `case 2:` to execute every time the `case 1:` statement executes.

d) Error: An improper relational operator is used in the `while`'s continuation condition.
Correction: Use `<=` rather than `<`, or change `10` to `11`.

Exercises

5.5 Describe the four basic elements of counter-controlled repetition.

5.6 Compare and contrast the `while` and `for` repetition statements.

5.7 Discuss a situation in which it would be more appropriate to use a `do…while` statement than a `while` statement. Explain why.

5.8 Compare and contrast the `break` and `continue` statements.

5.9 Find and correct the error(s) in each of the following segments of code:

a)
```
For ( i = 100, i >= 1, i++ )
   System.out.println( i );
```

b) The following code should print whether integer `value` is odd or even:

```
switch ( value % 2 )
{
   case 0:
      System.out.println( "Even integer" );

   case 1:
      System.out.println( "Odd integer" );
}
```

c) The following code should output the odd integers from 19 to 1:

```
for ( i = 19; i >= 1; i += 2 )
   System.out.println( i );
```

d) The following code should output the even integers from 2 to 100:

```
counter = 2;

do
{
   System.out.println( counter );
   counter += 2;
} While ( counter < 100 );
```

5.10 What does the following program do?

```java
// Exercise 5.10: Printing.java
public class Printing
{
   public static void main( String[] args )
   {
      for ( int i = 1; i <= 10; i++ )
      {
         for ( int j = 1; j <= 5; j++ )
            System.out.print( '@' );

         System.out.println();
      } // end outer for
   } // end main
} // end class Printing
```

5.11 *(Find the Smallest Value)* Write an application that finds the smallest of several integers. Assume that the first value read specifies the number of values to input from the user.

5.12 *(Calculating the Product of Odd Integers)* Write an application that calculates the product of the odd integers from 1 to 15.

5.13 *(Factorials) Factorials* are used frequently in probability problems. The factorial of a positive integer *n* (written *n*! and pronounced "*n* factorial") is equal to the product of the positive integers from 1 to *n*. Write an application that calculates the factorials of 1 through 20. Use type `long`. Display the results in tabular format. What difficulty might prevent you from calculating the factorial of 100?

5.14 *(Modified Compound-Interest Program)* Modify the compound-interest application of Fig. 5.6 to repeat its steps for interest rates of 5%, 6%, 7%, 8%, 9% and 10%. Use a `for` loop to vary the interest rate.

5.15 *(Triangle Printing Program)* Write an application that displays the following patterns separately, one below the other. Use `for` loops to generate the patterns. All asterisks (*) should be printed by a single statement of the form `System.out.print( '*' );` which causes the asterisks to print side by side. A statement of the form `System.out.println();` can be used to move to the next line. A statement of the form `System.out.print( ' ' );` can be used to display a space for the last two patterns. There should be no other output statements in the program. [*Hint:* The last two patterns require that each line begin with an appropriate number of blank spaces.]

```
(a)            (b)            (c)            (d)
*              **********     **********              *
**             *********       *********             **
***            ********         ********            ***
****           *******           *******           ****
*****          ******             ******          *****
******         *****               *****         ******
*******        ****                 ****        *******
********       ***                   ***       ********
*********      **                     **      *********
**********     *                       *     **********
```

5.16 *(Bar Chart Printing Program)* One interesting application of computers is to display graphs and bar charts. Write an application that reads five numbers between 1 and 30. For each number that's read, your program should display the same number of adjacent asterisks. For example, if your program reads the number 7, it should display `*******`. Display the bars of asterisks *after* you read all five numbers.

5.17 *(Calculating Sales)* An online retailer sells five products whose retail prices are as follows: Product 1, $2.98; product 2, $4.50; product 3, $9.98; product 4, $4.49 and product 5, $6.87. Write an application that reads a series of pairs of numbers as follows:

a) product number
b) quantity sold

Your program should use a `switch` statement to determine the retail price for each product. It should calculate and display the total retail value of all products sold. Use a sentinel-controlled loop to determine when the program should stop looping and display the final results.

5.18 *(Modified Compound-Interest Program)* Modify the application in Fig. 5.6 to use only integers to calculate the compound interest. [*Hint:* Treat all monetary amounts as integral numbers of pennies. Then break the result into its dollars and cents portions by using the division and remainder operations, respectively. Insert a period between the dollars and the cents portions.]

5.19 Assume that `i = 1`, `j = 2`, `k = 3` and `m = 2`. What does each of the following statements print?

a) `System.out.println( i == 1 );`
b) `System.out.println( j == 3 );`
c) `System.out.println( ( i >= 1 ) && ( j < 4 ) );`
d) `System.out.println( ( m <= 99 ) & ( k < m ) );`
e) `System.out.println( ( j >= i ) || ( k == m ) );`
f) `System.out.println( ( k + m < j ) | ( 3 - j >= k ) );`
g) `System.out.println( !( k > m ) );`

5.20 *(Calculating the Value of π)* Calculate the value of π from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Print a table that shows the value of π approximated by computing the first 200,000 terms of this series. How many terms do you have to use before you first get a value that begins with 3.14159?

5.21 *(Pythagorean Triples)* A right triangle can have sides whose lengths are all integers. The set of three integer values for the lengths of the sides of a right triangle is called a Pythagorean triple. The lengths of the three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Write an application that displays a table of the Pythagorean triples for `side1`, `side2` and the `hypotenuse`, all no larger than 500. Use a triple-nested `for` loop that tries all possibilities. This method is an example of "brute-force" computing. You'll learn in more advanced computer science courses that for many interesting problems there's no known algorithmic approach other than using sheer brute force.

5.22 *(Modified Triangle Printing Program)* Modify Exercise 5.15 to combine your code from the four separate triangles of asterisks such that all four patterns print side by side. [*Hint:* Make clever use of nested `for` loops.]

5.23 *(De Morgan's Laws)* In this chapter, we discussed the logical operators `&&`, `&`, `||`, `|`, `^` and `!`. De Morgan's laws can sometimes make it more convenient for us to express a logical expression. These laws state that the expression `!(`*condition1* `&&` *condition2*`)` is logically equivalent to the expression `(!`*condition1* `||` `!`*condition2*`)`. Also, the expression `!(`*condition1* `||` *condition2*`)` is logically equivalent to the expression `(!`*condition1* `&&` `!`*condition2*`)`. Use De Morgan's laws to write equivalent expressions for each of the following, then write an application to show that both the original expression and the new expression in each case produce the same value:

a) `!( x < 5 ) && !( y >= 7 )`
b) `!( a == b ) || !( g != 5 )`
c) `!( ( x <= 8 ) && ( y > 4 ) )`
d) `!( ( i > 4 ) || ( j <= 6 ) )`

5.24 *(Diamond Printing Program)* Write an application that prints the following diamond shape. You may use output statements that print a single asterisk (*), a single space or a single newline character. Maximize your use of repetition (with nested for statements), and minimize the number of output statements.

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

5.25 *(Modified Diamond Printing Program)* Modify the application you wrote in Exercise 5.24 to read an odd number in the range 1 to 19 to specify the number of rows in the diamond. Your program should then display a diamond of the appropriate size.

5.26 A criticism of the `break` statement and the `continue` statement is that each is unstructured. Actually, these statements can always be replaced by structured statements, although doing so can be awkward. Describe in general how you'd remove any `break` statement from a loop in a program and replace it with some structured equivalent. [*Hint:* The `break` statement exits a loop from the body of the loop. The other way to exit is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates "early exit because of a 'break' condition."] Use the technique you develop here to remove the `break` statement from the application in Fig. 5.12.

5.27 What does the following program segment do?

```java
for ( i = 1; i <= 5; i++ )
{
   for ( j = 1; j <= 3; j++ )
   {
      for ( k = 1; k <= 4; k++ )
         System.out.print( '*' );

      System.out.println();
   } // end inner for

   System.out.println();
} // end outer for
```

5.28 Describe in general how you'd remove any `continue` statement from a loop in a program and replace it with some structured equivalent. Use the technique you develop here to remove the `continue` statement from the program in Fig. 5.13.

5.29 *("The Twelve Days of Christmas" Song)* Write an application that uses repetition and `switch` statements to print the song "The Twelve Days of Christmas." One `switch` statement should be used to print the day ("first," "second," and so on). A separate `switch` statement should be used to print the remainder of each verse. Visit the website `en.wikipedia.org/wiki/The_Twelve_Days_of_Christmas_(song)` for the lyrics of the song.

Making a Difference

5.30 *(Global Warming Facts Quiz)* The controversial issue of global warming has been widely publicized by the film "An Inconvenient Truth," featuring former Vice President Al Gore. Mr. Gore and a U.N. network of scientists, the Intergovernmental Panel on Climate Change, shared the 2007 Nobel Peace Prize in recognition of "their efforts to build up and disseminate greater knowledge about man-made climate change." Research *both* sides of the global warming issue online (you might want to search for phrases like "global warming skeptics"). Create a five-question multiple-choice quiz on global warming, each question having four possible answers (numbered 1–4). Be objective and try to fairly represent both sides of the issue. Next, write an application that administers the quiz, calculates the number of correct answers (zero through five) and returns a message to the user. If the user correctly answers five questions, print "Excellent"; if four, print "Very good"; if three or fewer, print "Time to brush up on your knowledge of global warming," and include a list of some of the websites where you found your facts.

5.31 *(Tax Plan Alternatives; The "FairTax")* There are many proposals to make taxation fairer. Check out the FairTax initiative in the United States at

```
www.fairtax.org/site/PageSer PageServer?pagename=calculator
```

Research how the proposed FairTax works. One suggestion is to eliminate income taxes and most other taxes in favor of a 23% consumption tax on all products and services that you buy. Some FairTax opponents question the 23% figure and say that because of the way the tax is calculated, it would be more accurate to say the rate is 30%—check this carefully. Write a program that prompts the user to enter expenses in various expense categories they have (e.g., housing, food, clothing, transportation, education, health care, vacations), then prints the estimated FairTax that person would pay.

5.32 *(Facebook User Base Growth)* According to CNNMoney.com, Facebook hit 500 million users in July of 2010 and its user base has been growing at a rate of 5% per month. Using the compound-growth technique you learned in Fig. 5.6 and assuming this growth rate continues, how many months will it take for Facebook to grow its user base to one billion users? How many months will it take for Facebook to grow its user base to two billion users (which, at the time of this writing, was the total number of people on the Internet)?