3

Control Statements: Part 1; Assignment, ++ and -- Operators

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

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.
- Develop algorithms through the process of top-down, stepwise refinement.
- Use the `if` and `if...else` selection statements to choose between alternative actions.
- Use the `while` repetition statement to execute statements in a program repeatedly.
- Use counter-controlled repetition and sentinel-controlled repetition.
- Use the compound assignment operator, and the increment and decrement operators.
- Learn about the portability of primitive data types.

Outline

3.1 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 the next, we discuss these issues in presenting the theory and principles of structured programming. As you'll see when we get into object-oriented programming (starting in Chapter 7), the concepts presented here are crucial in constructing classes and manipulating objects. We discuss Java's `if` statement in additional detail, and introduce the `if...else` and `while` statements—all of these building blocks allow you to specify the logic required for methods to perform their tasks. We also introduce the compound assignment operator and the increment and decrement operators. Finally, we consider the portability of Java's primitive types.

3.2 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**.

3.3 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. The pseudocode we use in this book 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. 3.5. You may, of course, use your own native language(s) to develop your own pseudocode.

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*. In our pseudocode, we typically do *not* include variable declarations, but some programmers choose to list variables and mention their purposes.

3.4 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. [*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." The term **structured programming** became almost synonymous with "`goto` elimination." 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.

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

1. C. Bohm, 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.

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. 3.1 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.

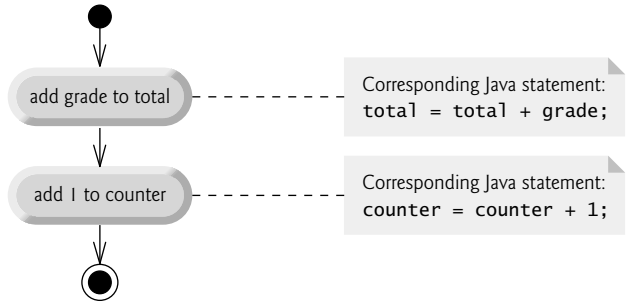

Fig. 3.1 | Sequence-structure activity diagram.

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. 3.1. 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. Activity diagrams clearly show how control structures operate. We use the UML in this chapter and Chapter 4 to show control flow in control statements. Online Chapters 33–34 use the UML in a real-world automated-teller-machine case study.

Consider the sequence-structure activity diagram in Fig. 3.1. It contains two **action states**, each containing 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. 3.1 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** at the bottom of the diagram represents the **final state**—the *end* of the workflow *after* the program performs its actions.

Figure 3.1 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 sym-

bols in the diagram. Figure 3.1 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 online object-oriented design case study (Chapters 33–34) or visit `www.uml.org`.

Selection Statements in Java

Java has three types of **selection statements** (discussed in this chapter and Chapter 4). 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 4) 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 **iteration statements** or **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`, `for` and enhanced `for` statements. (Chapter 4 presents the `do...while` and `for` statements and Chapter 6 presents the enhanced `for` statement.) 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. 3.1, 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.

3.5 `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.

UML Activity Diagram for an `if` Statement

Figure 3.2 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. 3.2, 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.

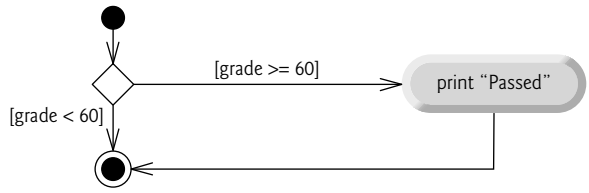

Fig. 3.2 | `if` single-selection statement UML activity diagram.

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.

3.6 `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 another 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

```
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 3.1

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

Good Programming Practice 3.2

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

UML Activity Diagram for an `if…else` Statement

Figure 3.3 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.

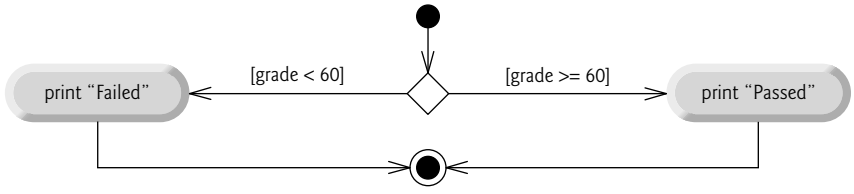

Fig. 3.3 | `if…else` double-selection statement UML activity diagram.

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");
```

Error-Prevention Tip 3.1

In a nested `if...else` *statement, ensure that you test for all possible cases.*

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 3.273.28 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 (such as the body of a

method) form a **block**. A block can be placed anywhere in a method 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 3.1

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).

Conditional Operator (`?:`)

Java provides the **conditional operator** (**`?:`**) that can be used in place of an `if...else` statement. This can make your code shorter and clearer. The conditional operator is Java's only **ternary operator** (i.e., an 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.

Error-Prevention Tip 3.2

Use expressions of the same type for the second and third operands of the `?:` operator to avoid subtle errors.

3.7 `while` Repetition Statement

A repetition 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 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 shopping list's last item 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;
```

Each iteration of the `while` statement multiplies `product` by 3, so `product` takes on the values 9, 27, 81 and 243 successively. When `product` becomes 243, `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 3.2

*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).*

UML Activity Diagram for a `while` Statement

The UML activity diagram in Fig. 3.4 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.

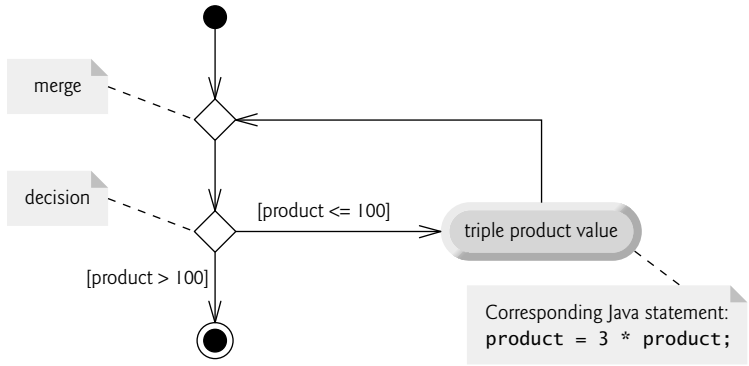

Fig. 3.4 | `while` repetition statement UML activity diagram.

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.

Figure 3.4 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.

3.8 Formulating Algorithms: Counter-Controlled Repetition

To illustrate how algorithms are developed, we 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 0–100) 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. 3.5) and a corresponding Java program (Fig. 3.6) that implements the algorithm. In Section 3.9, we demonstrate how to use pseudocode to develop such an algorithm from scratch.

Note the references in the algorithm of Fig. 3.5 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.

Software Engineering Observation 3.1

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 it is usually straightforward.

```
1  Set total to zero
2  Set grade counter to one
3
4  While grade counter is less than or equal to ten
5      Prompt the user to enter the next grade
6      Input the next grade
7      Add the grade into the total
8      Add one to the grade counter
9
10 Set the class average to the total divided by ten
11 Print the class average
```

Fig. 3.5 | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

Implementing Counter-Controlled Repetition

In Fig. 3.6, class `ClassAverage`'s `main` method (lines 7–31) implements the class-averaging algorithm described by the pseudocode in Fig. 3.5—it allows the user to enter 10 grades, then calculates and displays the average.

```
1  // Fig. 3.6: ClassAverage.java
2  // Solving the class-average problem using counter-controlled repetition.
3  import java.util.Scanner; // program uses class Scanner
4
```

Fig. 3.6 | Solving the class-average problem using counter-controlled repetition. (Part 1 of 2.)

```
 5  public class ClassAverage
 6  {
 7     public static void main(String[] args)
 8     {
 9        // create Scanner to obtain input from command window
10        Scanner input = new Scanner(System.in);
11
12        // initialization phase
13        int total = 0;  // initialize sum of grades entered by the user
14        int gradeCounter = 1; // initialize # of grade to be entered next
15
16        // processing phase uses counter-controlled repetition
17        while (gradeCounter <= 10) // loop 10 times
18        {
19           System.out.print("Enter grade: "); // prompt
20           int grade = input.nextInt(); // input next grade
21           total = total + grade; // add grade to total
22           gradeCounter = gradeCounter + 1; // increment counter by 1
23        }
24
25        // termination phase
26        int average = total / 10; // integer division yields integer result
27
28        // display total and average of grades
29        System.out.printf("%nTotal of all 10 grades is %d%n", total);
30        System.out.printf("Class average is %d%n", average);
31     }
32  } // end class ClassAverage
```

```
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. 3.6 | Solving the class-average problem using counter-controlled repetition. (Part 2 of 2.)

Local Variables in Method `main`

Line 10 declares and initializes `Scanner` object `input`, which is used to read values entered by the user. Lines 13, 14, 20 and 26 declare local variables `total`, `gradeCounter`, `grade` and `average`, respectively, to be of type `int`. Variable `grade` stores the user input.

These declarations appear in the body of method `main`. 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 out-

side the method in which it's declared. Variable `grade`, declared in the body of the `while` loop, can be used only in that block.

Initialization Phase: Initializing Variables `total` *and* `gradeCounter`

The assignments (in lines 13–14) initialize `total` to `0` and `gradeCounter` to `1`. These initializations occur *before* the variables are used in calculations.

Common Programming Error 3.3

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 3.3

Initialize each total and counter, 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).

Processing Phase: Reading 10 Grades from the User

Line 17 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 18–23).

Line 19 displays the prompt `"Enter grade: "`. Line 20 reads the grade entered by the user and assigns it to variable `grade`. Then line 21 adds the new `grade` entered by the user to the `total` and assigns the result to `total`, which replaces its previous value.

Line 22 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 17) becomes *false*.

Termination Phase: Calculating and Displaying the Class Average

When the loop terminates, line 26 performs the averaging calculation and assigns its result to the variable `average`. Line 29 uses `System.out`'s `printf` method to display the text `"Total of all 10 grades is "` followed by variable `total`'s value. Line 30 then uses `printf` to display the text `"Class average is "` followed by variable `average`'s value. When execution reaches line 31, the program terminates.

Notes on Integer Division and Truncation

The averaging calculation performed by method `main` 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 26 of Fig. 3.6) 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 **truncated** (i.e., *lost*). In the next section we'll see how to obtain a floating-point result from the averaging calculation.

Common Programming Error 3.4

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.

A Note About Arithmetic Overflow

In Fig. 3.6, line 21

```
total = total + grade; // add grade to total
```

added each `grade` entered by the user to the `total`. Even this simple statement has a *potential* problem—adding the integers could result in a value that's *too large* to store in an `int` variable. This is known as **arithmetic overflow** and causes *undefined behavior*, which can lead to unintended results (`http://en.wikipedia.org/wiki/Integer_overflow#Security_ramifications`). Figure 2.7's `Addition` program had the same issue in line 23, which calculated the sum of two `int` values entered by the user:

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

The maximum and minimum values that can be stored in an `int` variable are represented by the constants `MIN_VALUE` and `MAX_VALUE`, respectively, which are defined in class `Integer`. There are similar constants for the other integral types and for floating-point types. Each primitive type has a corresponding class type in package `java.lang`. You can see the values of these constants in each class's online documentation. The online documentation for class Integer is located at:

```
http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html
```

It's considered a good practice to ensure, *before* you perform arithmetic calculations like those in line 21 of Fig. 3.6 and line 23 of Fig. 2.7, that they will *not* overflow. The code for doing this is shown on the CERT website `www.securecoding.cert.org`—just search for guideline "NUM00-J." The code uses the `&&` (logical AND) and `||` (logical OR) operators, which are introduced in Chapter 4. In industrial-strength code, you should perform checks like these for *all* calculations.

A Deeper Look at Receiving User Input

Any time a program receives input from the user, various problems might occur. For example, in line 20 of Fig. 3.6

```
int grade = input.nextInt(); // input next grade
```

we assume that the user will enter an integer grade in the range 0 to 100. However, the person entering a grade could enter an integer less than 0, an integer greater than 100, an integer outside the range of values that can be stored in an `int` variable, a number containing a decimal point or a value containing letters or special symbols that's not even an integer.

To ensure that inputs are valid, industrial-strength programs must test for all possible erroneous cases. A program that inputs grades should **validate** the grades by using **range checking** to ensure that hey are values from 0 to 100. You can then ask the user to reenter any value that's out of range. If a program requires inputs from a specific set of values (e.g., nonsequential product codes), you can ensure that each input matches a value in the set.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

Let's generalize Section 3.8'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 3.2

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

Software Engineering Observation 3.3

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 *repetition* to successively input each grade. We do not know in advance how many grades will be entered, 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. 3.7.

Error-Prevention Tip 3.4

When performing division (/) or remainder (%) calculations in which the right operand could be zero, test for this and handle it (e.g., display an error message) rather than allowing the error to occur.

```
1  Initialize total to zero
2  Initialize counter to zero
3
4  Prompt the user to enter the first grade
5  Input the first grade (possibly the sentinel)
6
7  While the user has not yet entered the sentinel
8      Add this grade into the running total
9      Add one to the grade counter
10     Prompt the user to enter the next grade
11     Input the next grade (possibly the sentinel)
12
13 If the counter is not equal to zero
14     Set the average to the total divided by the counter
15     Print the average
16 else
17     Print "No grades were entered"
```

Fig. 3.7 | Class-average pseudocode algorithm with sentinel-controlled repetition.

In Figs. 3.5 and 3.7, 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. 3.7 solves the more general class-average problem. This algorithm was developed after two refinements. Sometimes more are needed.

Software Engineering Observation 3.4

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 3.5

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 Fig. 3.8, method `main` (lines 7–46) implements the pseudocode algorithm of Fig. 3.7. Although each grade is an `int`, the averaging calculation is likely to produce a number with a *decimal point*—in other words, a real (floating-point) number. The type `int` cannot represent such a number, so this class uses type `double` to do so. You'll also see that control statements may be *stacked* on top of one another (in sequence). The `while` statement (lines 22–30) is followed in sequence by an `if...else` statement (lines 34–45). Much of the code in this program is identical to that in Fig. 3.6, so we concentrate on the new concepts.

```
1  // Fig. 3.8: ClassAverage.java
2  // Solving the class-average problem using sentinel-controlled repetition.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class ClassAverage
6  {
7     public static void main(String[] args)
8     {
9        // create Scanner to obtain input from command window
10       Scanner input = new Scanner(System.in);
11
12       // initialization phase
13       int total = 0; // initialize sum of grades
14       int gradeCounter = 0; // initialize # of grades entered so far
15
16       // processing phase
17       // prompt for input and read grade from user
18       System.out.print("Enter grade or -1 to quit: ");
19       int grade = input.nextInt();
20
21       // loop until sentinel value read from user
22       while (grade != -1)
23       {
24          total = total + grade; // add grade to total
25          gradeCounter = gradeCounter + 1; // increment counter
26
27          // prompt for input and read next grade from user
28          System.out.print("Enter grade or -1 to quit: ");
29          grade = input.nextInt();
30       }
31
32       // termination phase
33       // if user entered at least one grade...
34       if (gradeCounter != 0)
35       {
36          // use number with decimal point to calculate average of grades
37          double average = (double) total / gradeCounter;
38
39          // display total and average (with two digits of precision)
40          System.out.printf("%nTotal of the %d grades entered is %d%n",
41             gradeCounter, total);
```

Fig. 3.8 | Solving the class-average problem using sentinel-controlled repetition. (Part 1 of 2.)

```
42            System.out.printf("Class average is %.2f%n", average);
43         }
44         else // no grades were entered, so output appropriate message
45            System.out.println("No grades were entered");
46      }
47   } // end class ClassAverage
```

```
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. 3.8 | Solving the class-average problem using sentinel-controlled repetition. (Part 2 of 2.)

Floating-Point Number Precision and Memory Requirements

Most averages are not integers. So, this example calculates the class average as a **floating-point number**—a number with a *decimal point*, such as 43.95, 0.0, –129.8873. Java provides two primitive types for storing floating-point numbers in memory—`float` and `double`. Variables of type **`float`** represent **single-precision floating-point numbers** and can hold 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 can hold up to *15 significant digits*—about *double* the precision of `float` variables.

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 precise ranges of values for `float`s and `double`s.

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

Line 37 declares `double` variable `average`, which allows us to store the class average as a floating-point number. Line 14 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.

Compare the program logic for sentinel-controlled repetition in this application with that for counter-controlled repetition in Fig. 3.6. In counter-controlled repetition, each iteration of the `while` statement (lines 17–23 of Fig. 3.6) reads a value from the user, for the specified number of iterations. In sentinel-controlled repetition, the program reads the first value (lines 18–19 of Fig. 3.8) 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` and increments the `gradeCounter` (lines 24–25). Then lines 28–29 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 30,

so execution continues with the test of the `while`'s condition (line 22). 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 3.3

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

After the loop terminates, the `if...else` statement at lines 34–45 executes. The condition at line 34 determines whether any grades were input. If none were input, the `else` part (lines 44–45) of the `if...else` statement executes and displays the message `"No grades were entered"` and the method returns control to the calling method.

Braces in a `while` statement

Notice the `while` statement's *block* in Fig. 3.8 (lines 23–30). 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 19 (before the `while` statement).

Common Programming Error 3.5

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 37 of Fig. 3.8 calculates the average of the grades. Recall from Fig. 3.6 that integer division yields an integer result. Even though variable `average` is declared as a `double`, if we had written the averaging calculation as

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

it would lose 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.

Most averages are not whole numbers (e.g., 0, –22 and 1024). For this reason, we calculate the class average in this example as a floating-point number. 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 37 of Fig. 3.8 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` copy of `total`) divided by the integer `gradeCounter`. Java can evaluate only arithmetic expressions in which the operands' types are *identical*. To ensure this, Java performs an operation called **promotion** (or **implicit conversion**) on selected operands. For example, in an expression containing `int` and `double` values, 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 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 5.7.

Common Programming Error 3.6

A cast operator can be used to convert between primitive numeric types, such as `int` *and* `double`*. Casting to the wrong type may cause compilation 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 42 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 (Fig. 3.8) 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`.

Floating-Point Number Precision

Floating-point numbers are not always 100% precise, but 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.

Floating-point numbers often arise as a result of division, such as in this example's class-average calculation. 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.

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. In some applications, the precision of `float` and `double` variables will be inadequate. For precise floating-point numbers (such as those required by monetary calculations), Java provides class `BigDecimal` (package `java.math`), which we'll discuss in Chapter 8.

Common Programming Error 3.7

Using floating-point numbers in a manner that assumes they're represented precisely can lead to incorrect results.

3.10 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 "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 3.24 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. 3.9. 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.

```
1  Initialize passes to zero
2  Initialize failures to zero
3  Initialize student counter to one
4
5  While student counter is less than or equal to 10
6      Prompt the user to enter the next exam result
7      Input the next exam result
8
9      If the student passed
10         Add one to passes
11     Else
12         Add one to failures
13
14     Add one to student counter
15
16 Print the number of passes
17 Print the number of failures
18
19 If more than eight students passed
20     Print "Bonus to instructor!"
```

Fig. 3.9 | Pseudocode for examination-results problem.

The Java class that implements the pseudocode algorithm and two sample executions are shown in Fig. 3.10. Lines 13, 14, 15 and 22 of `main` declare the variables that are used to process the examination results.

Error-Prevention Tip 3.5
Initializing local variables when they're declared helps you avoid 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 each local variable be given a value before its value is used in an expression.

The `while` statement (lines 18–32) loops 10 times. During each iteration, the loop inputs and processes one exam result. Notice that the `if...else` statement (lines 25–28) 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 31 increments `studentCounter` before the loop condition is tested again at line 18. After 10 values have been input, the loop terminates and line 35 displays the number of `passes` and `failures`. The `if` statement at lines 38–39 determines whether more than eight students passed the exam and, if so, outputs the message `"Bonus to instructor!"`.

```
1  // Fig. 3.10: Analysis.java
2  // Analysis of examination results using nested control statements.
3  import java.util.Scanner; // class uses class Scanner
4
5  public class Analysis
6  {
7     public static void main(String[] args)
8     {
9        // create Scanner to obtain input from command window
10       Scanner input = new Scanner(System.in);
11
12       // initializing variables in declarations
13       int passes = 0;
14       int failures = 0;
15       int studentCounter = 1;
16
17       // process 10 students using counter-controlled loop
18       while (studentCounter <= 10)
19       {
20          // prompt user for input and obtain value from user
21          System.out.print("Enter result (1 = pass, 2 = fail): ");
22          int result = input.nextInt();
23
24          // if...else is nested in the while statement
25          if (result == 1)
26             passes = passes + 1;
27          else
28             failures = failures + 1;
29
30          // increment studentCounter so loop eventually terminates
31          studentCounter = studentCounter + 1;
32       }
33
34       // termination phase; prepare and display results
35       System.out.printf("Passed: %d%nFailed: %d%n", passes, failures);
36
37       // determine whether more than 8 students passed
38       if (passes > 8)
39          System.out.println("Bonus to instructor!");
40    }
41 } // end class Analysis
```

Fig. 3.10 | Analysis of examination results using nested control statements. (Part 1 of 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): 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!
```

```
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. 3.10 | Analysis of examination results using nested control statements. (Part 2 of 2.)

Figure 3.10 shows the input and output from two sample excutions of the program. During the first, the condition at line 38 of method `main` is `true`—more than eight students passed the exam, so the program outputs a message to bonus the instructor.

3.11 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 3.11 shows the arithmetic compound assignment operators, sample expressions using the operators and explanations of what the operators do.

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. 3.11 | Arithmetic compound assignment operators.

3.12 Increment and Decrement Operators

Java provides two unary operators (summarized in Fig. 3.12) 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 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. 3.12 | 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 3.4

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

Difference Between Prefix Increment and Postfix Increment Operators

Figure 3.13 demonstrates the difference between the prefix increment and postfix increment versions of the ++ increment operator. The decrement operator (--) works similarly.

Line 9 initializes the variable c to 5, and line 10 outputs c's initial value. Line 11 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 11 outputs c's initial value (5) again. Line 12 outputs c's new value (6) to prove that the variable's value was indeed incremented in line 11.

Line 17 resets c's value to 5, and line 18 outputs c's value. Line 19 outputs the value of the expression ++c. This expression preincrements c, so its value is incremented; then the *new* value (6) is output. Line 20 outputs c's value again to show that the value of c is still 6 after line 19 executes.

```
1  // Fig. 3.13: Increment.java
2  // Prefix increment and postfix increment operators.
3
4  public class Increment
5  {
6     public static void main(String[] args)
7     {
8        // demonstrate postfix increment operator
9        int c = 5;
10       System.out.printf("c before postincrement: %d%n", c); // prints 5
11       System.out.printf("    postincrementing c: %d%n", c++); // prints 5
12       System.out.printf(" c after postincrement: %d%n", c); // prints 6
13
14       System.out.println(); // skip a line
15
16       // demonstrate prefix increment operator
17       c = 5;
18       System.out.printf(" c before preincrement: %d%n", c); // prints 5
19       System.out.printf("     preincrementing c: %d%n", ++c); // prints 6
20       System.out.printf("  c after preincrement: %d%n", c); // prints 6
21    }
22 } // end class Increment
```

```
c before postincrement: 5
    postincrementing c: 5
 c after postincrement: 6

 c before preincrement: 5
     preincrementing c: 6
  c after preincrement: 6
```

Fig. 3.13 | Prefix increment and postfix increment operators.

Simplifying Statements with the Arithmetic Compound Assignment, Increment and Decrement Operators

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. 3.10 (lines 26, 28 and 31)

```
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 3.8

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.

Operator Precedence and Associativity

Figure 3.14 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. 3.14 associate from left to right. The third column lists the type of each group of operators.

Good Programming Practice 3.5

Refer to Appendix A, Operator Precedence Chart, when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you're uncertain about the evaluation order, break the expression into smaller statements or use parentheses to force the evaluation order, exactly as you'd do in an algebraic expression. Some operators such as assignment (=) associate right to left rather than left to right.

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. 3.14 | Precedence and associativity of the operators discussed so far.

3.13 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` on one machine might be represented by 16 bits (2 bytes) of memory, on a second machine by 32 bits (4 bytes), and on another machine by 64 bits (8 bytes). In Java, `int` values are always 32 bits (4 bytes).

Portability Tip 3.1

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/`).

3.14 (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. 3.15), a scheme for identifying points on the screen. By default, the upper-left corner of a GUI component 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 hor-

izontal 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.

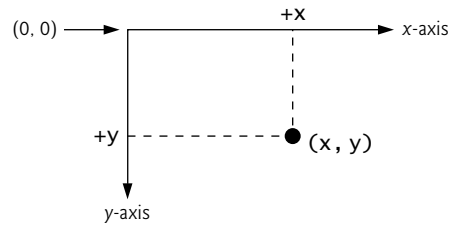

Fig. 3.15 | Java coordinate system. Units are measured in pixels.

First Drawing Application

Our first drawing application simply draws two lines. Class `DrawPanel` (Fig. 3.16) performs the actual drawing, while class `DrawPanelTest` (Fig. 3.17) 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. 3.16: 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    }
23 } // end class DrawPanel
```

Fig. 3.16 | Using `drawLine` to connect the corners of a panel.

```
1  // Fig. 3.17: DrawPanelTest.java
2  // Creating JFrame to display DrawPanel.
3  import javax.swing.JFrame;
4
5  public class DrawPanelTest
6  {
7     public static void main(String[] args)
8     {
9        // create a panel that contains our drawing
10       DrawPanel panel = new DrawPanel();
11
12       // create a new frame to hold the panel
13       JFrame application = new JFrame();
14
15       // set the frame to exit when it is closed
16       application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17
18       application.add(panel); // add the panel to the frame
19       application.setSize(250, 250); // set the size of the frame
20       application.setVisible(true); // make the frame visible
21    }
22 } // end class DrawPanelTest
```

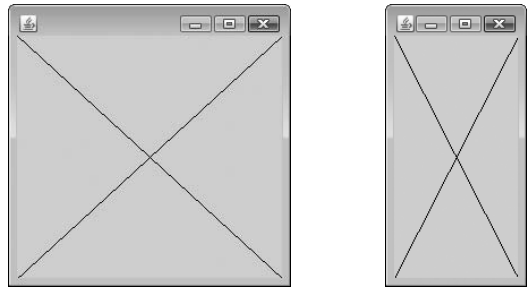

Fig. 3.17 | Creating `JFrame` to display `DrawPanel`.

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 `DrawPanel`. 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`. This `Graphics` object is used to draw lines, rectangles, ovals and other graphics.

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

```
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. 3.17), 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. 3.16) is implicitly called, and the two lines are drawn (see the sample outputs in Fig. 3.17). 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

3.1 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. 3.18. 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. 3.18. Lines from opposite corners should intersect along the middle.

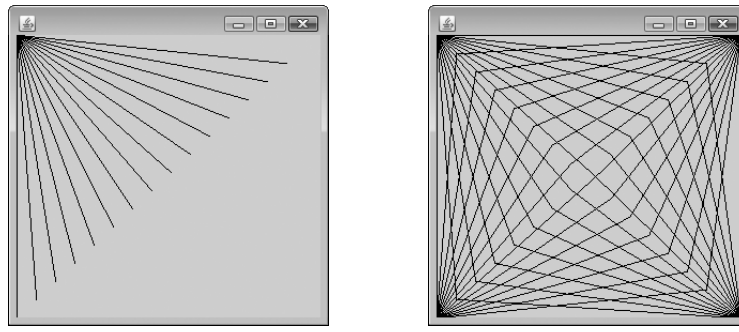

Fig. 3.18 | Lines fanning from a corner.

3.2 Figure 3.19 displays two additional designs created using `while` loops and `drawLine`.

a) Create the design in the left screen capture of Fig. 3.19. 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. 3.19.

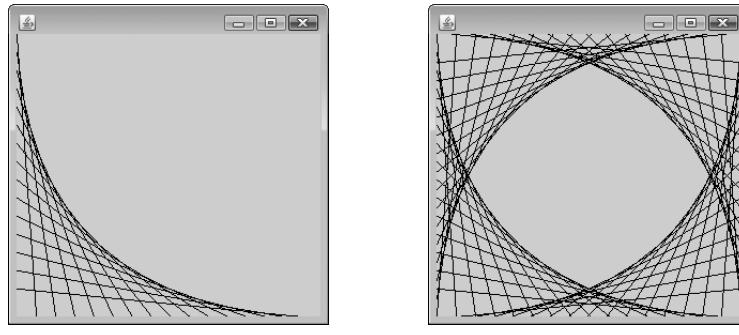

Fig. 3.19 | Line art with loops and `drawLine`.

3.15 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 4, we continue our discussion of control statements, introducing the `for`, `do...while` and `switch` statements.

Summary

Section 3.1 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 3.2 Algorithms

- Any computing problem can be solved by executing a series of actions (p. 73) 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. 73).
- Specifying the order in which statements execute in a program is called program control (p. 73).

Section 3.3 Pseudocode

- Pseudocode (p. 74) is an informal language that helps you develop algorithms without having to worry about the strict details of Java language syntax.
- The pseudocode we use in this book is similar to everyday English—it's convenient and user friendly, but it's not an actual computer programming language. You may, of course, use your own native language(s) to develop your own pseudocode.
- 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 3.4 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. 74).
- 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. 74).
- Bohm and Jacopini demonstrated that all programs could be written in terms of only three control structures (p. 74)—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. 75).

- 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. 75) are part of the UML. An activity diagram models the workflow (p. 75; also called the activity) of a portion of a software system.
- Activity diagrams are composed of symbols (p. 75)—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. 75) 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. 75)—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. 75)—the end of the workflow after the program performs its actions.
- Rectangles with their upper-right corners folded over are UML notes (p. 75)—explanatory remarks that describe the purpose of symbols in the diagram.
- Java has three types of selection statements (p. 76).
- The `if` single-selection statement (p. 76) 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. 76) because it selects among many different actions or groups of actions.
- Java provides the `while`, `do...while` and `for` repetition (also called iteration or 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. 76) 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. 76) as is appropriate for the algorithm the program implements.
- Single-entry/single-exit control statements (p. 76) 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. 76) inside another control statement.

Section 3.5 `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. 77). 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 3.6 `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. 76) statement performs one action when the condition is true and another action when the condition is false.
- A program can test multiple cases with nested `if...else` statements (p. 78).
- The conditional operator (`?:`, p. 81) is Java's only ternary operator—it takes three operands. Together, the operands and the `?:` symbol form a conditional expression (p. 81).
- 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. 81) of statements can be placed anywhere that a single statement can be placed.
- A logic error (p. 81) has its effect at execution time. A fatal logic error (p. 81) causes a program to fail and terminate prematurely. A nonfatal logic error (p. 81) 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, you can also use an empty statement, represented by placing a semicolon (`;`) where a statement would normally be.

Section 3.7 `while` Repetition Statement

- The `while` repetition statement (p. 82) allows you to specify that a program should repeat an action while some condition remains true.
- The UML's merge (p. 82) 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 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 3.8 Formulating Algorithms: Counter-Controlled Repetition

- Counter-controlled repetition (p. 84) 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. 84), because the number of repetitions is known before the loop begins executing.
- A total (p. 84) 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—the calculation's fractional part is truncated.

Section 3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- In sentinel-controlled repetition (p. 88), 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. 88) 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 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 on selected operands.
- The unary cast operator is formed by placing parentheses around the name of a type.

Section 3.11 Compound Assignment Operators

- The compound assignment operators (p. 99) 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 3.12 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. 100).
- An increment or decrement operator that's prefixed (p. 100) to a variable is the prefix increment or prefix decrement operator, respectively. An increment or decrement operator that's postfixed (p. 100) 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 3.13 Primitive Types

- Java requires all variables to have a type. Thus, Java is referred to as a strongly typed language (p. 103).
- Java uses Unicode characters and IEEE 754 floating-point numbers.

Self-Review Exercises

3.1 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.

3.2 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.

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

3.4 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.

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

3.6 Combine the statements that you wrote in Exercise 3.5 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.

3.7 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.

3.8 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");
```

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

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

Answers to Self-Review Exercises

3.1 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.

3.2 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.

3.3

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

3.4 a)

```
z = x++ + y;
```

b)

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

c)

```
total -= --x;
```

d)

```
q %= divisor;
```

or

```
q = q % divisor;
```

3.5 a)

```
int sum = 0;
```

b)

```
int x = 1;
```

c)

```
sum += x;
```

or

```
sum = sum + x;
```

d)

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

3.6 The program is as follows:

```
// Exercise 3.6: Calculate.java
// Calculate the sum of the integers from 1 to 10
public class Calculate
{
   public static void main(String[] args)
   {
      int sum = 0;
      int x = 1;

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

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

```
The sum is: 55
```

3.7 `product = 25, x = 6`

3.8 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`.

3.9 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

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

3.11 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?

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

3.13 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.

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

3.15 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;
```

3.16 What does the following program print?

```
1  // Exercise 3.16: Mystery.java
2  public class Mystery
3  {
4     public static void main(String[] args)
5     {
6        int x = 1;
7        int total = 0;
```

```
 8
 9         while (x <= 10)
10         {
11            int y = x * x;
12            System.out.println(y);
13            total += y;
14            ++x;
15         }
16
17         System.out.printf("Total is %d%n", total);
18      }
19   } // end class Mystery
```

For Exercises 3.17 through 3.20, 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.

3.17 *(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.

3.18 *(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"`.

3.19 *(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 $5,000 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.

3.20 *(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.

3.21 *(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.

3.22 *(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
```

3.23 *(Find the Two Largest Numbers)* Using an approach similar to that for Exercise 3.21, find the *two* largest values of the 10 values entered. [*Note:* You may input each number only once.]

3.24 *(Validating User Input)* Modify the program in Fig. 3.10 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.

3.25 What does the following program print?

```
1  // Exercise 3.25: Mystery2.java
2  public class Mystery2
3  {
4     public static void main(String[] args)
5     {
6        int count = 1;
7
8        while (count <= 10)
9        {
10          System.out.println(count % 2 == 1 ? "****" : "++++++++");
11          ++count;
12       }
13    }
14 } // end class Mystery2
```

3.26 What does the following program print?

```
1  // Exercise 3.26: Mystery3.java
2  public class Mystery3
3  {
4     public static void main(String[] args)
5     {
```

```
 6         int row = 10;
 7
 8         while (row >= 1)
 9         {
10            int column = 1;
11
12            while (column <= 10)
13            {
14               System.out.print(row % 2 == 1 ? "<" : ">");
15               ++column;
16            }
17
18            --row;
19            System.out.println();
20         }
21      }
22   } // end class Mystery3
```

3.27 ***(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("$$$$$");
}
```

3.28 ***(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.]

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

3.29 ***(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.

3.30 ***(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.

3.31 ***(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.]

3.32 ***(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.]

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

3.33 ***(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?

3.34 *(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));
```

3.35 *(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.

3.36 *(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.

3.37 *(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

$$n! = 1 \quad \text{(for } n = 0\text{)}$$

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

3.38 *(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.]

3.39 *(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 sometime 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.