

4

Control Structures: Part 1

Objectives

- To understand basic problem-solving techniques.
- To be able to develop algorithms through the process of top-down, stepwise refinement.
- To be able to use the **if** and **if/else** selection structures to choose among alternative actions.
- To be able to use the **while** repetition structure to execute statements in a program repeatedly.
- To understand counter-controlled repetition and sentinel-controlled repetition.
- To be able to use the increment, decrement and assignment operators.

Let's all move one place on.

Lewis Carroll

The wheel is come full circle.

William Shakespeare, *King Lear*

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

Robert Frost, comment



Outline

- 4.1 Introduction
- 4.2 Algorithms
- 4.3 Pseudocode
- 4.4 Control Structures
- 4.5 The `if` Selection Structure
- 4.6 The `if/else` Selection Structure
- 4.7 The `while` Repetition Structure
- 4.8 Formulating Algorithms: Case Study 1 (Counter-Controlled Repetition)
- 4.9 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 2 (Sentinel-Controlled Repetition)
- 4.10 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 3 (Nested Control Structures)
- 4.11 Assignment Operators
- 4.12 Increment and Decrement Operators
- 4.13 Primitive Data Types
- 4.14 (Optional Case Study) Thinking About Objects: Identifying Class Attributes

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

4.1 Introduction

Before writing a program to solve a problem, it is essential to have a thorough understanding of the problem and a carefully planned approach to solving the problem. When writing a program, it is equally essential to understand the types of building blocks that are available and to employ proven program construction principles. In this chapter and in Chapter 5, we discuss these issues in our presentation of the theory and principles of structured programming. The techniques you learn here are applicable to most high-level languages, including Java. When we study object-based programming in more depth in Chapter 8, we will see that control structures are helpful in building and manipulating objects.

4.2 Algorithms

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

1. the *actions* to be executed and
2. the *order* in which the actions are to be executed

is called an *algorithm*. The following example demonstrates that correctly specifying the order in which the actions are to be executed is important.

Consider the “rise-and-shine algorithm” followed by one junior 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, however, 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 junior executive shows up for work soaking wet. Specifying the order in which statements are to be executed in a computer program is called *program control*. In this chapter and Chapter 5, we investigate the program control capabilities of Java.

4.3 Pseudocode

Pseudocode is an artificial and informal language that helps programmers develop algorithms. The pseudocode we present here is particularly useful for developing algorithms that will be converted to structured portions of Java programs. Pseudocode is similar to everyday English; it is convenient and user friendly, although it is not an actual computer programming language.

Pseudocode programs are not actually executed on computers. Rather, they help the programmer “think out” a program before attempting to write it in a programming language, such as Java. In this chapter, we give several examples of pseudocode programs.



Software Engineering Observation 4.1

Pseudocode is often used to “think out” a program during the program design process. Then the pseudocode program is converted to Java.

The style of pseudocode we present consists purely of characters, so programmers may conveniently type pseudocode programs using an editor program. The computer can produce a freshly printed copy of a pseudocode program on demand. A carefully prepared pseudocode program may be converted easily to a corresponding Java program. This conversion is done in many cases simply by replacing pseudocode statements with their Java equivalents.

Pseudocode normally describes only executable statements—the actions that are performed when the program is converted from pseudocode to Java and is run. Declarations are not executable statements. For example, the declaration

```
int i;
```

tells the compiler the type of variable **i** and instructs the compiler to reserve space in memory for the variable. This declaration does not cause any action—such as input, output or a calculation—to occur when the program is executed. Some programmers choose to list variables and mention the purpose of each at the beginning of a pseudocode program.

4.4 Control Structures

Normally, statements in a program are executed one after the other in the order in which they are written. This process is called *sequential execution*. Various Java statements we will soon discuss enable the programmer to specify that the next statement to be executed may be other than 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 finger of blame was pointed at the **goto** statement (used in several programming languages, including C and Basic), which allows the programmer to specify a transfer of control to one of a very wide range of possible destinations in a program. The notion of so-called *structured programming* became almost synonymous with “**goto** elimination.” Java does not have a **goto** statement; however, **goto** is a reserved word and should not be used in a Java program.

The research of Bohm and Jacopini¹ 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.” It was not until the 1970s that programmers started taking structured programming seriously. The results have been impressive, as software development groups have reported reduced development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes is that structured programs are 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*—namely, the *sequence structure*, the *selection structure* and the *repetition structure*. 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 are written. The *flowchart* segment in Fig. 4.1 illustrates a typical sequence structure in which two calculations are performed in order.

A flowchart is a graphical representation of an algorithm or a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles; these symbols are connected by arrows called *flowlines*, which indicate the order in which the actions of the algorithm execute.

Like pseudocode, flowcharts are often useful for developing and representing algorithms, although pseudocode is strongly preferred by many programmers. Flowcharts show clearly how control structures operate; that is all we use them for in this text. The reader should carefully compare the pseudocode and flowchart representations of each control structure.

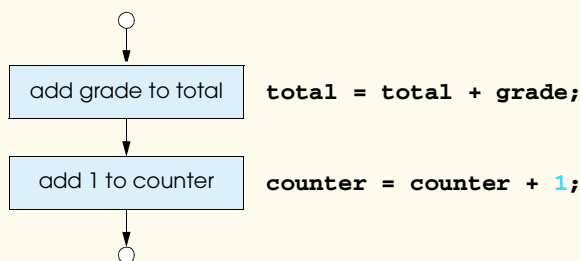


Fig. 4.1 Flowcharting Java’s sequence structure.

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

Consider the flowchart segment for the sequence structure in Fig. 4.1. We use the *rectangle symbol*, also called the *action symbol*, to indicate any type of action, including a calculation or an input/output operation. The flowlines in the figure indicate the order in which the actions are to be performed; first, **grade** is to be added to **total**, and then **1** is to be added to **counter**. Java allows us to have as many actions as we want in a sequence structure. As we will soon see, anywhere a single action may be placed, we may instead place several actions in sequence.

When drawing a flowchart that represents a *complete* algorithm, an *oval symbol* containing the word “Begin” is the first symbol used in the flowchart; an oval symbol containing the word “End” indicates where the algorithm ends. When drawing only a portion of an algorithm, as in Fig. 4.1, the oval symbols are omitted in favor of *small circle symbols*, also called *connector symbols*.

Perhaps the most important flowcharting symbol is the *diamond symbol*, also called the *decision symbol*, which indicates that a decision is to be made. We will discuss the diamond symbol in the next section.

Java provides three types of selection structures; we discuss each in this chapter and in Chapter 5. The **if** selection structure either performs (selects) an action, if a condition is true, or skips the action, if the condition is false. The **if/else** selection structure performs an action if a condition is true and performs a different action if the condition is false. The **switch** selection structure (Chapter 5) performs one of many different actions, depending on the value of an expression.

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

Java provides three types of repetition structures—namely, **while**, **do/while** and **for**. (**do/while** and **for** are covered in Chapter 5.) Each of the words **if**, **else**, **switch**, **while**, **do** and **for** are Java *keywords*. These words are reserved by the language to implement various features, such as Java’s control structures. Keywords cannot be used as identifiers, such as for variable names. A complete list of Java keywords is shown in Fig. 4.2.

Java Keywords				
abstract	boolean	break	byte	case
catch	char	class	continue	default
do	double	else	extends	false
final	finally	float	for	if
implements	import	instanceof	int	interface
long	native	new	null	package
private	protected	public	return	short

Fig. 4.2 Java keywords (part 1 of 2).

Java Keywords				
static	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		
<i>Keywords that are reserved, but not used, by Java</i>				
const	goto			

Fig. 4.2 Java keywords (part 2 of 2).



Common Programming Error 4.1

Using a keyword as an identifier is a syntax error.

Well, that is all there is. Java has only seven control structures: the sequence structure, three types of selection structures and three types of repetition structures. Each program is formed by combining as many of each type of control structure as is appropriate for the algorithm the program implements. As with the sequence structure in Fig. 4.1, we will see that each control structure is flowcharted with two small circle symbols, one at the entry point to the control structure and one at the exit point.

Single-entry/single-exit control structures make it easy to build programs; the control structures are attached to one another by connecting the exit point of one control structure to the entry point of the next. This procedure is similar to the way in which a child stacks building blocks, so we call it *control-structure stacking*. We will learn that there is only one other way in which control structures may be connected: *control-structure nesting*. Thus, algorithms in Java programs are constructed from only seven different types of control structures, combined in only two ways.

4.5 The `if` Selection Structure

A selection structure is used to choose among alternative courses of action in a program. For example, suppose that the passing grade on an examination is 60 (out of 100). Then the pseudocode statement

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

determines if the condition “student’s grade is greater than or equal to 60” is true or false. If the condition is true, then “Passed” is printed, and the next pseudocode statement in order is “performed.” (Remember that 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. Note that the second line of this selection structure is indented. Such indentation is optional, but it is highly recommended, because it emphasizes the inherent structure of structured programs. The Java compiler ignores white-space characters, like blanks, tabs and newlines, used for indentation and vertical spacing. Programmers insert these white-space characters to enhance program clarity.

**Good Programming Practice 4.1**

Consistently applying reasonable indentation conventions throughout your programs improves program readability. We suggest a fixed-size tab of about 1/4 inch or three spaces per indent.

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

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

Notice that the Java code corresponds closely to the pseudocode. This attribute is a property of pseudocode that makes it a useful program development tool. The statement in the body of the **if** structure outputs the character string **"Passed"** in the command window.

The flowchart in Fig. 4.3 illustrates the single-selection **if** structure. This flowchart contains what is perhaps the most important flowcharting symbol—the *diamond symbol*, also called the *decision symbol*, which indicates that a decision is to be made. The decision symbol contains an expression, such as a condition, that can be either **true** or **false**. The decision symbol has two flowlines emerging from it. One indicates the direction to be taken when the expression in the symbol is true; the other indicates the direction to be taken when the expression is false. A decision can be made on any expression that evaluates to a value of Java's **boolean** type (i.e., any expression that evaluates to **true** or **false**).

Note that the **if** structure is a single-entry/single-exit structure. We will soon learn that the flowcharts for the remaining control structures also contain (besides small circle symbols and flowlines) only rectangle symbols, to indicate the actions to be performed, and diamond symbols, to indicate decisions to be made. This factor is indicative of the *action/decision model of programming* we have been emphasizing throughout this chapter.

We can envision seven bins, each containing only control structures of one of the seven types. These control structures are empty; nothing is written in the rectangles or in the diamonds. The programmer's task, then, is to assemble a program from as many of each type of control structure as the algorithm demands, combining the control structures in only two possible ways (stacking or nesting) and then filling in the actions and decisions in a manner appropriate for the algorithm. In this chapter we discuss the variety of ways in which actions and decisions may be written.

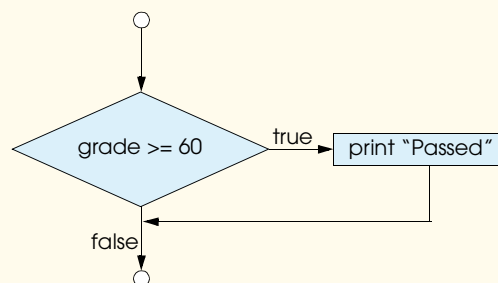


Fig. 4.3 Flowcharting the single-selection **if** structure.

4.6 The **if/else** Selection Structure

The **if** selection structure performs an indicated action only when the given condition evaluates to **true**; otherwise, the action is skipped. The **if/else** selection structure allows the programmer to specify that a different action is to be performed when the condition is true rather than 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 and prints *Failed* if the student's grade is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is "performed." Note that the body of the *else* is also indented.



Good Programming Practice 4.2

*Indent both body statements of an **if/else** structure.*

The indentation convention you choose should be carefully applied throughout your programs. It is difficult to read programs that do not use uniform spacing conventions.

The preceding pseudocode *If/else* structure may be written in Java as

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

The flowchart in Fig. 4.4 nicely illustrates the flow of control in an **if/else** structure. Once again, note that, besides small circles and arrows, the only symbols in the flowchart are rectangles (for actions) and a diamond (for a decision). We continue to emphasize this action/decision model of computing. Imagine again a deep bin containing as many empty double-selection structures as might be needed to build a Java algorithm. The programmer's job is to assemble the selection structures (by stacking and nesting) with other control structures required by the algorithm and to fill in the empty rectangles and empty diamonds with actions and decisions appropriate to the algorithm being implemented.

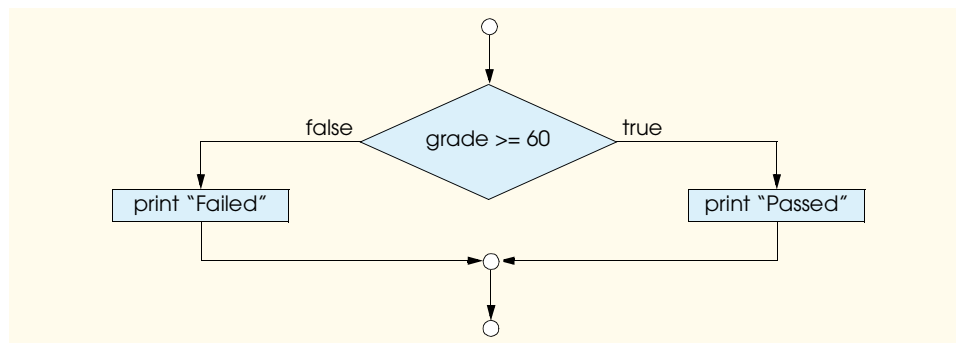


Fig. 4.4 Flowcharting the double-selection **if/else** structure.

The *conditional operator* (`?:`) is related to the **if/else** structure. `?:` is Java's only *ternary operator*—it takes three operands. The operands together with `?:` form a *conditional expression*. The first operand is a **boolean** expression, the second is the value for the conditional expression if the condition evaluates to **true** and the third is the value for the conditional expression if the condition evaluates to **false**. For example, the statement

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

contains a conditional expression that evaluates to the string **"Passed"** if the condition **studentGrade >= 60** is true and to the string **"Failed"** if the condition is false. Thus, this statement with the conditional operator performs essentially the same function as the **if/else** statement given previously. The precedence of the conditional operator is low, so the entire conditional expression is normally placed in parentheses. We will see that conditional operators can be used in some situations where **if/else** statements cannot.



Good Programming Practice 4.3

*In general, conditional expressions are more difficult to read than **if/else** structures. Such expressions should be used with discretion when they help improve a program's readability.*

*Nested **if/else** structures* test for multiple cases by placing **if/else** structures inside **if/else** structures. For example, the following pseudocode statement prints **A** for exam grades greater than or equal to 90, **B** for grades in the range 80 to 89, **C** for grades in the range 70 to 79, **D** for grades in the range 60 to 69 and **F** for all other grades:

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

This pseudocode may be written in Java as

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

If **studentGrade** is greater than or equal to 90, the first four conditions will be true, but only the **System.out.println** statement after the first test will be executed. After that particular **System.out.println** is executed, the **else** part of the “outer” **if/else** statement is skipped.



Good Programming Practice 4.4

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

Most Java programmers prefer to write the preceding **if** structure as

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

Both forms are equivalent. The latter form is popular because it avoids the deep indentation of the code to the right. Such deep indentation often leaves little room on a line, forcing lines to be split and decreasing program readability.

It is important to note that the Java compiler always associates an **else** with the previous **if** unless told to do otherwise by the placement of braces (**{}**). This attribute 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 **if** structure in its body determines if **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** structure outputs the string "**x is <= 5**".

Beware! The preceding nested **if** structure does not execute as it would appear to. The compiler actually interprets the preceding structure 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** structure is an **if/else** structure. This structure tests if **x** is greater than 5. If so, execution continues by testing if **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.

To force the preceding nested **if** structure to execute as it was originally intended, the structure must be written 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 to the compiler that the second **if** structure is in the body of the first **if** structure and that the **else** is matched with the first **if** structure. In Exercise 4.21 and Exercise 4.22, you will investigate the dangling-else problem further.

The **if** selection structure normally expects only one statement in its body. To include several statements in the body of an **if** structure, enclose the statements in braces (**{** and **}**). A set of statements contained within a pair of braces is called a *block*.



Software Engineering Observation 4.2

A block can be placed anywhere in a program that a single statement can be placed.

The following example includes a block in the **else** part of an **if/else** structure:

```
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.
```

Notice 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** structure and would execute regardless of whether the grade is less than 60.



Common Programming Error 4.2

Forgetting one or both of the braces that delimit a block can lead to syntax or logic errors.

Syntax errors (such as when one brace in a block is left out of the program) are caught by the compiler. A *logic error* (such as 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 the program produces incorrect results.



Software Engineering Observation 4.3

Just as a block can be placed anywhere a single statement can be placed, it is also possible to have no statement at all (i.e., the empty statement in such places). The empty statement is represented by placing a semicolon (;) where a statement would normally be.



Common Programming Error 4.3

*Placing a semicolon after the condition in an **if** structure leads to a logic error in single-selection **if** structures and a syntax error in double-selection **if** structures (if the **if** part contains a nonempty body statement).*



Good Programming Practice 4.5

Some programmers prefer to type the beginning and ending braces of blocks before typing the individual statements within the braces. This practice helps avoid omitting one or both of the braces.

In this section, we have introduced the notion of a block. A block may contain declarations (as does the body of **main**, for example). The declarations in a block commonly are placed first in the block before any action statements occur, but declarations may also be intermixed with action statements.

4.7 The **while** Repetition Structure

A *repetition structure* allows the programmer to specify that an action is to be repeated while some condition remains true. The pseudocode statement

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

describes the repetition that occurs during a shopping trip. The condition “there are more items on my shopping list” may be true or false. If it is 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 structure constitute the body of the *while* structure. The body of the *while* structure may be a single statement or a block. Eventually, the condition will become false (when the last item on the shopping list has been purchased and crossed off the list). At this point, the repetition terminates, and the first pseudocode statement after the repetition structure is executed.



Common Programming Error 4.4

*Not providing in the body of a **while** structure an action that eventually causes the condition in the **while** to become false is a logic error. Normally, such a repetition structure will never terminate—an error called an infinite loop.*



Common Programming Error 4.5

*Spelling the keyword **while** with an uppercase **W**, as in **While**, is a syntax error. (Remember that Java is a case-sensitive language.) All of Java’s reserved keywords, such as **while**, **if** and **else**, contain only lowercase letters.*

As an example of a **while** structure, consider a program segment designed to find the first power of 2 larger than 1000. Suppose that the **int** variable **product** has been initialized to 2. When the following **while** structure finishes executing, **product** contains the result:

```
int product = 2;

while ( product <= 1000 )
    product = 2 * product;
```

The flowchart in Fig. 4.5 illustrates the flow of control of the preceding **while** repetition structure. Once again, note that, besides small circles and arrows, the flowchart contains only a rectangle symbol and a diamond symbol.

Imagine, again, a deep bin of empty **while** structures that may be stacked and nested with other control structures to form a structured implementation of an algorithm's flow of control. The empty rectangles and diamonds are then filled in with appropriate actions and decisions. The flowchart clearly shows the repetition. The flowline emerging from the rectangle wraps back to the decision, which is tested each time through the loop until the decision eventually becomes false. At this point, the **while** structure is exited, and control passes to the next statement in the program.

When the **while** structure is entered, **product** is 2. Variable **product** is repeatedly multiplied by 2, taking on the values 4, 8, 16, 32, 64, 128, 256, 512 and 1024 successively. When **product** becomes 1024, the condition **product <= 1000** in the **while** structure becomes **false**. This result terminates the repetition, with 1024 as **product**'s final value. Execution continues with the next statement after the **while**. [Note: If a **while** structure's condition is initially **false**, the body statement(s) will never be performed.]

4.8 Formulating Algorithms: Case Study 1 (Counter-Controlled Repetition)

To illustrate how algorithms are developed, we solve several variations of a class-averaging problem. Consider the following problem statement:

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

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

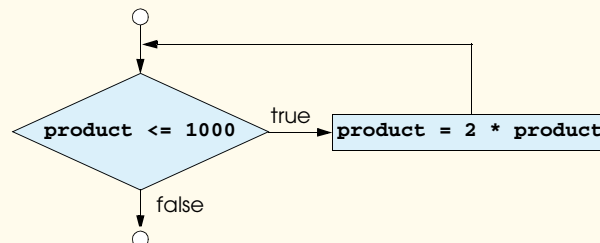


Fig. 4.5 Flowcharting the **while** repetition structure.

Let us use pseudocode to list the actions to be executed and specify the order in which these actions should be executed. We use *counter-controlled repetition* to input the grades one at a time. This technique uses a variable called a *counter* to control the number of times a set of statements will execute. In this example, repetition terminates when the counter exceeds 10. In this section, we present a pseudocode algorithm (Fig. 4.6) and the corresponding program (Fig. 4.7) to solve this problem using counter-controlled repetition. In the next section, we show how pseudocode algorithms are developed. Counter-controlled repetition is often called *definite repetition*, because the number of repetitions is known before the loop begins executing.

Note the references in the algorithm to a total and a counter. A *total* is a variable used to accumulate the sum of a series of values. A *counter* is a variable used to count—in this case, to count the number of grades entered. Variables used to store totals should normally be initialized to zero before being used in a program; otherwise, the sum would include the previous value stored in the total's memory location.

Set total to zero
Set grade counter to one

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

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

Fig. 4.6 Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

```

1  // Fig. 4.7: Averagel.java
2  // Class average program with counter-controlled repetition.
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class Averagel {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         int total,           // sum of grades input by user
13             gradeCounter,   // number of grades entered
14             gradeValue,     // grade value
15             average;        // average of all grades
16         String grade;       // grade typed by user
17     }

```

Fig. 4.7 Class-average program with counter-controlled repetition (part 1 of 3).

```

18      // Initialization Phase
19      total = 0;           // clear total
20      gradeCounter = 1;    // prepare to loop
21
22      // Processing Phase
23      while ( gradeCounter <= 10 ) { // loop 10 times
24
25          // prompt for input and read grade from user
26          grade = JOptionPane.showInputDialog(
27              "Enter integer grade: " );
28
29          // convert grade from a String to an integer
30          gradeValue = Integer.parseInt( grade );
31
32          // add gradeValue to total
33          total = total + gradeValue;
34
35          // add 1 to gradeCounter
36          gradeCounter = gradeCounter + 1;
37
38      } // end while structure
39
40      // Termination Phase
41      average = total / 10; // perform integer division
42
43      // display average of exam grades
44      JOptionPane.showMessageDialog( null,
45          "Class average is " + average, "Class Average",
46          JOptionPane.INFORMATION_MESSAGE );
47
48      System.exit( 0 ); // terminate the program
49
50  } // end method main
51
52  } // end class Average1

```

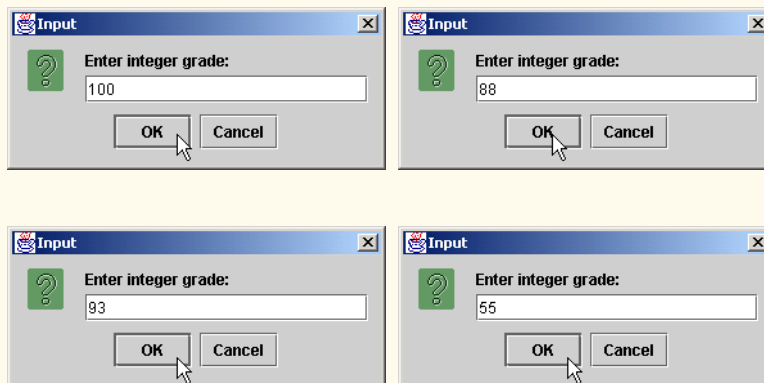


Fig. 4.7 Class-average program with counter-controlled repetition (part 2 of 3).

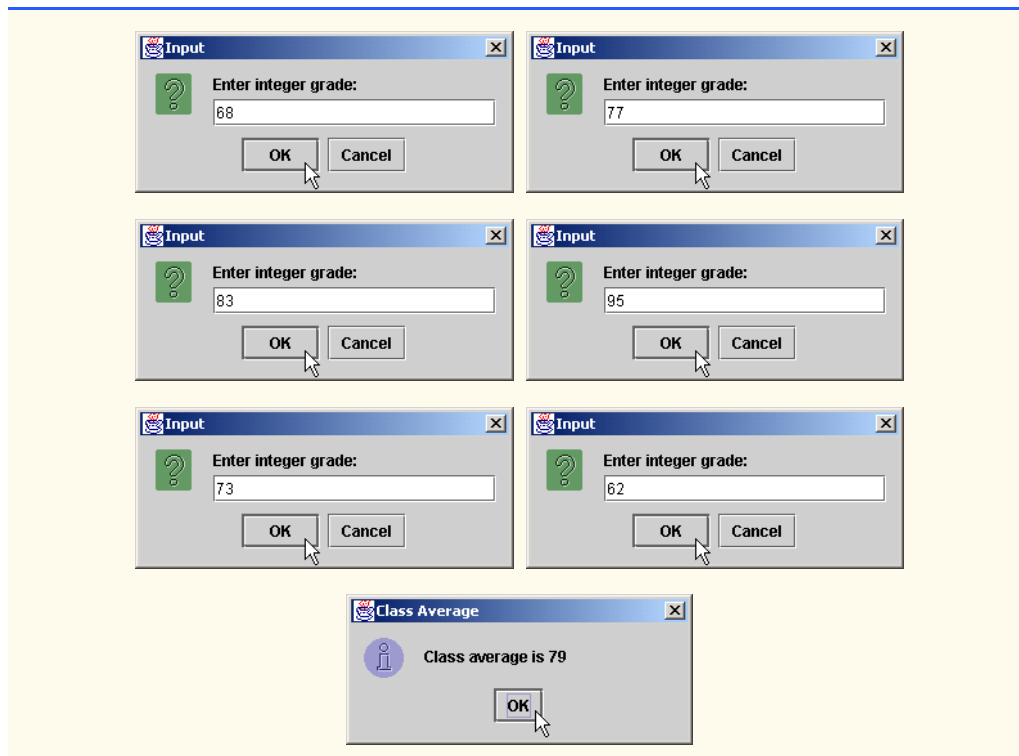


Fig. 4.7 Class-average program with counter-controlled repetition (part 3 of 3).



Good Programming Practice 4.6

Initialize counters and totals.

Line 5,

```
import javax.swing.JOptionPane;
```

imports class **JOptionPane** to enable the program to read data from the keyboard and output data to the screen using the input dialog and message dialog shown in Chapter 2.

Line 7 begins the definition of application class **Average1**. Remember that the definition of an application class must contain a **main** method (lines 10–49) in order for the application to be executed.

Lines 12–16,

```
int total,           // sum of grades
    gradeCounter,    // number of grades entered
    gradeValue,      // grade value
    average;         // average of all grades
String grade;        // grade typed by user
```

declare variables **total**, **gradeCounter**, **gradeValue** and **average** to be of type **int** and variable **grade** to be of type **String**. Variable **grade** stores the **String** the

user types in the input dialog. Variable **gradeValue** stores the integer value of **grade** after the program converts it from a **String** to an **int**.

Notice that the preceding declarations appear in the body of method **main**. Remember that variables declared in a method definition's body are *local variables* and can be used only from the line of their declaration in the method to the closing right brace (**}**) of the method definition. A local variable's declaration must appear before the variable is used in that method. A local variable declared in one method of a class cannot be accessed directly by other methods of a class.



Good Programming Practice 4.7

Always place a blank line before a declaration that appears between executable statements. This format makes the declarations stand out in the program and contributes to program clarity.



Good Programming Practice 4.8

If you prefer to place declarations at the beginning of a method, separate the declarations from the executable statements in that method with one blank line, to highlight where the declarations end and the executable statements begin.



Common Programming Error 4.6

Attempting to use a local variable's value before initializing the variable (normally with an assignment statement) results in a compile error indicating that the variable may not have been initialized. The value of a local variable cannot be used until the variable is initialized. The program will not compile properly until the variable receives an initial value.

Lines 19–20,

```
total = 0;           // clear total
gradeCounter = 1;    // prepare to loop
```

are assignment statements that initialize **total** to 0 and **gradeCounter** to 1. Note that these statements initialize variables **total** and **gradeCounter** before they are used in calculations.

Line 23,

```
while ( gradeCounter <= 10 ) { // loop 10 times
```

indicates that the **while** structure should continue looping (also called *iterating*) as long as the value of **gradeCounter** is less than or equal to 10.

Lines 26–27,

```
grade = JOptionPane.showInputDialog(
    "Enter integer grade: " );
```

correspond to the pseudocode statement “*Input the next grade.*” The statement displays an input dialog with the prompt “**Enter integer grade:**” on the screen.

After the user enters the **grade**, the program converts it from a **String** to an **int** at line 30,

```
gradeValue = Integer.parseInt( grade );
```

Remember that class **Integer** is from package **java.lang** that the compiler imports in every Java program. The pseudocode for the class-average problem does not reflect the pre-

ceding statement. The pseudocode statement “*Input the next grade*” requires the programmer to implement the process of obtaining the value from the user and converting it to a type that can be used in calculating the average. As you learn to program, you will find that you require fewer pseudocode statements to help you implement a program.

Next, the program updates the **total** with the new **gradeValue** entered by the user. Line 33,

```
total = total + gradeValue;
```

adds **gradeValue** to the previous value of **total** and assigns the result to **total**.

Line 36,

```
gradeCounter = gradeCounter + 1;
```

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** is necessary for the condition in the **while** structure to become **false** eventually and terminate the loop.

Line 41,

```
average = total / 10; // perform integer division
```

assigns the results of the average calculation to variable **average**. Lines 44–46,

```
JOptionPane.showMessageDialog(  
    null, "Class average is " + average, "Class Average",  
    JOptionPane.INFORMATION_MESSAGE );
```

display an information message dialog containing the string “**Class average is** ” followed by the value of variable **average**. The string “**Class Average**” (the third argument) is the title of the message dialog.

Line 48,

```
System.exit( 0 ); // terminate the program
```

terminates the application.

After compiling the class definition with **javac**, execute the application from the command window with the command

```
java Average1
```

This command executes the Java interpreter and tells it that the **main** method for this application is defined in class **Average1**.

Note that the averaging calculation in the program produced an integer result. Actually, the sum of the grade-point values in this example is 794, which, when divided by 10, should yield 79.4 (i.e., a number with a decimal point). We will see how to deal with such numbers (called *floating-point numbers*) in the next section.

4.9 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 2 (Sentinel-Controlled Repetition)

Let us generalize the class-average problem. Consider the following problem:

Develop a class-averaging program that processes an arbitrary number of grades each time the program executes.

In the first class-average example, the number of grades (10) was known in advance. In this example, no indication is given of how many grades the user will input. The program must process an arbitrary number of grades. How can the program 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 the end of data entry. The user types grades in until all legitimate grades have been entered. The user then types the sentinel value to indicate that the last grade has been entered. Sentinel-controlled repetition is often called *indefinite repetition*, because the number of repetitions is not known before the loop begins executing.

Clearly, the sentinel value must be chosen so that it cannot be confused with an acceptable input value. Because grades on a quiz are normally nonnegative integers, -1 is an acceptable sentinel value for this problem. Thus, an execution of the class-average program might process a stream of inputs such as 95, 96, 75, 74, 89 and -1 . In this case, the program would compute and print the class average for the grades 95, 96, 75, 74 and 89. (-1 is the sentinel value, so it should not enter into the averaging calculation.)



Common Programming Error 4.7

Choosing a sentinel value that is also a legitimate data value results in a logic error and may prevent a sentinel-controlled loop from terminating properly.

We approach the class-average program with a technique called *top-down, stepwise refinement*, a method that is essential to the development of well-structured algorithms. We begin with a pseudocode representation of the *top*:

Determine the class average for the quiz

The top is a single statement that conveys the overall function of the program. As such, the top is, in effect, a complete representation of a program. Unfortunately, the top rarely conveys a sufficient amount of detail from which to write the Java algorithm. So we now begin the refinement process. We divide the top into a series of smaller tasks and list these tasks in the order in which they need to be performed. This procedure results in the following *first refinement*:

Initialize variables

Input, sum up and count the quiz grades

Calculate and print the class average

This pseudocode uses only the sequence structure—the steps listed occur in order, one after the other.



Software Engineering Observation 4.4

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

To proceed to the next level of refinement (i.e., the *second refinement*), we commit to specific variables. We need a running total of the grades, a count of how many grades have been processed, a variable to receive the value of each grade as it is input and a variable to store the calculated average. The pseudocode statement

Initialize variables

may be refined as follows:

Initialize total to zero
Initialize counter to zero

Notice that only the variables *total* and *counter* are initialized before they are used; the variables *average* and *grade* (for the calculated average and the user input, respectively) need not be initialized, because their values are replaced as they are calculated or input.

The pseudocode statement

Input, sum up and count the quiz grades

requires a repetition structure (i.e., a loop) that successively inputs each grade. We do not know how many grades the user will input, so the program will use sentinel-controlled repetition. The user at the keyboard inputs legitimate grades one at a time. After inputting the last legitimate grade, the user types the sentinel value. The program tests for the sentinel value after each grade is input and terminates the loop when the user inputs the sentinel value. The second refinement of the preceding pseudocode statement is then

Input the first grade (possibly the sentinel)

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

Notice that in pseudocode, we do not use braces around the pseudocode that forms the body of the *while* structure. We simply indent the pseudocode under the *while*, to show that it belongs to the *while*. Again, pseudocode is only an informal program development aid.

The pseudocode statement

Calculate and print the class average

may 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"

Notice that we are testing for the possibility of division by zero—a *logic error* that, if undetected, would cause the program to produce invalid output. The complete second refinement of the pseudocode algorithm for the class-average problem is shown in Fig. 4.8.



Testing and Debugging Tip 4.1

When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program (such as by printing an error message) rather than allowing the division by zero to occur.



Good Programming Practice 4.9

Include completely blank lines in pseudocode programs to make the pseudocode more readable. The blank lines separate pseudocode control structures, as well as the phases of the programs.

```

Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)

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

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

```

Fig. 4.8 Pseudocode algorithm that uses sentinel-controlled repetition to solve the class-average problem.



Software Engineering Observation 4.5

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

The pseudocode algorithm in Fig. 4.8 solves the more general class-averaging problem. This algorithm was developed after only two levels of refinement. Sometimes more levels are necessary.



Software Engineering Observation 4.6

The programmer terminates the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for the programmer to be able to convert the pseudocode to a Java applet or application. Normally, implementing the Java applet or application is then straightforward.

The Java application and a sample execution are shown in Fig. 4.9. Although each grade is an integer, the averaging calculation is likely to produce a number with a decimal point (i.e., a real number). The type **int** cannot represent real numbers (i.e., numbers with decimal points), so this program uses data type **double** to handle floating-point numbers. The program introduces a special operator called a *cast operator* to handle the type conversion we will need for the averaging calculation. These features are explained in detail in the discussion of the application.

In this example, we see that control structures may be stacked on top of one another (in sequence) just as a child stacks building blocks. The **while** structure (lines 33–47) is followed by an **if/else** structure (lines 52–63) in sequence. Much of the code in this program is identical to the code in Fig. 4.7, so we concentrate in this example on the new features and issues.

```
1  // Fig. 4.9: Average2.java
2  // Class average program with sentinel-controlled repetition.
3
4  // Java core packages
5  import java.text.DecimalFormat;
6
7  // Java extension packages
8  import javax.swing.JOptionPane;
9
10 public class Average2 {
11
12     // main method begins execution of Java application
13     public static void main( String args[] )
14     {
15         int gradeCounter,    // number of grades entered
16             gradeValue,      // grade value
17             total;           // sum of grades
18         double average;      // average of all grades
19         String input;        // grade typed by user
20
21         // Initialization phase
22         total = 0;           // clear total
23         gradeCounter = 0;    // prepare to loop
24
25         // Processing phase
26         // prompt for input and read grade from user
27         input = JOptionPane.showInputDialog(
28             "Enter Integer Grade, -1 to Quit:" );
29
30         // convert grade from a String to an integer
31         gradeValue = Integer.parseInt( input );
32
33         while ( gradeValue != -1 ) {
34
35             // add gradeValue to total
36             total = total + gradeValue;
37
38             // add 1 to gradeCounter
39             gradeCounter = gradeCounter + 1;
40
41             // prompt for input and read grade from user
42             input = JOptionPane.showInputDialog(
43                 "Enter Integer Grade, -1 to Quit:" );
44
45             // convert grade from a String to an integer
46             gradeValue = Integer.parseInt( input );
47         }
48
49         // Termination phase
50         DecimalFormat twoDigits = new DecimalFormat( "0.00" );
51
52         if ( gradeCounter != 0 ) {
53             average = (double) total / gradeCounter;
```

Fig. 4.9 Class-average program with sentinel-controlled repetition (part 1 of 2).

```

54
55         // display average of exam grades
56         JOptionPane.showMessageDialog( null,
57             "Class average is " + twoDigits.format( average ),
58             "Class Average", JOptionPane.INFORMATION_MESSAGE );
59     }
60     else
61         JOptionPane.showMessageDialog( null,
62             "No grades were entered", "Class Average",
63             JOptionPane.INFORMATION_MESSAGE );
64
65     System.exit( 0 );    // terminate application
66
67 } // end method main
68
69 } // end class Average2

```

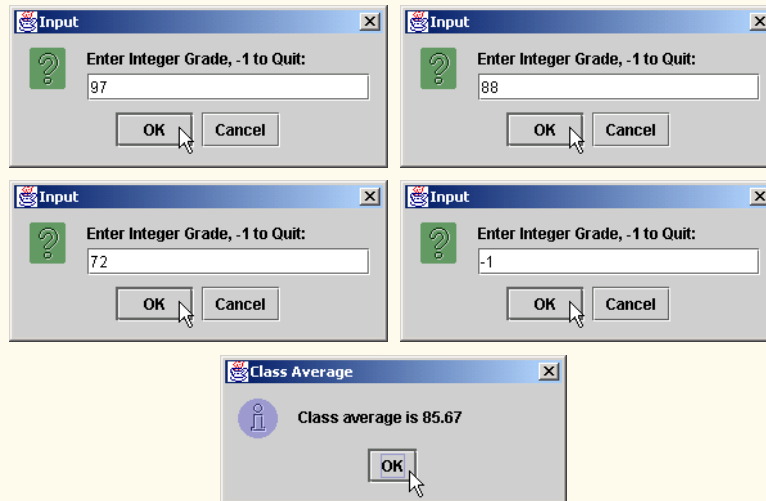


Fig. 4.9 Class-average program with sentinel-controlled repetition (part 2 of 2).

Line 18 declares **double** variable **average**. This change allows us to store the class average as a floating-point number. Line 23 initializes **gradeCounter** to 0, because no grades have been entered yet. Remember that this program uses sentinel-controlled repetition. To keep an accurate record of the number of grades entered, variable **gradeCounter** is incremented only when the user inputs a valid grade value.

Notice the difference in program logic for sentinel-controlled repetition as compared with the counter-controlled repetition in Fig. 4.7. In counter-controlled repetition, each iteration (loop) of the **while** structure reads a value from the user, for the specified number of iterations. In sentinel-controlled repetition, the program reads and converts one value (lines 27–31) before reaching the **while** structure. This value determines whether the program's flow of control should enter the body of the **while** structure. If the condition of the **while** structure is **false**, the user entered the sentinel, so the body of the **while** structure 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 value input by the user to the

total. After the value has been processed, lines 42–46 in the loop’s body input the next value from the user before program control reaches the end of the **while** structure’s body. As program control reaches the closing right brace (**}**) of the body at line 47, execution continues with the next test of the condition of the **while** structure (line 33). The condition uses the new value just input by the user to determine if the **while** structure’s body should execute again. Notice that the next value always is input from the user immediately before the program tests the condition of the **while** structure. This structure allows the program to determine if the value just input by the user is the sentinel value *before* the program processes that value (i.e., adds it to the **total**). If the value input is the sentinel value, the **while** structure terminates, and the program does not add the value to the **total**.

Notice the block in the **while** loop in Fig. 4.9. Without the braces, the last four statements in the body of the loop would fall outside the loop, causing the computer to interpret the code incorrectly as follows:

```
while ( gradeValue != -1 )
    // add gradeValue to total
    total = total + gradeValue;
// add 1 to gradeCounter
gradeCounter = gradeCounter + 1;
// prompt for input and read grade from user
input = JOptionPane.showInputDialog(
    "Enter Integer Grade, -1 to Quit:" );
// convert grade from a String to an integer
gradeValue = Integer.parseInt( input );
```

This code would cause an infinite loop in the program if the user does not input the sentinel **-1** as the input value at lines 27–28 (before the **while** structure) in the program.



Common Programming Error 4.8

Omitting the curly braces that are needed to delineate a block can lead to logic errors such as infinite loops. To prevent this problem, some programmers enclose the body of every control structure in braces.



Good Programming Practice 4.10

In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user of the value that represents the sentinel.

Line 50,

```
DecimalFormat twoDigits = new DecimalFormat( "0.00" );
```

declares **twoDigits** as a reference to an object of class **DecimalFormat** (package **java.text**). **DecimalFormat** objects format numbers. In this example, we want to output the class average with two digits to the right of the decimal point (i.e., rounded to the nearest hundredth). The preceding line creates a **DecimalFormat** object that is initialized with the string **"0.00"**. Each **0** is a *format flag* that specifies a required digit position in the formatted floating-point number. This particular format indicates that every number formatted with **twoDigits** will have at least one digit to the left of the decimal point and exactly two digits to the right of the decimal point. If the number does not meet the formatting requirements, **0**s are inserted in the formatted number at the required positions. The **new operator** creates an

object as the program executes by obtaining enough memory to store an object of the type specified to the right of **new**. The process of creating new objects is also known as *creating an instance*, or *instantiating an object*. Operator **new** is known as the *dynamic memory allocation operator*. The value in parentheses after the type in a **new** operation is used to *initialize* (i.e., give a value to) the new object. Reference **twoDigits** is assigned the result of the **new** operation by using the *assignment operator*, **=**. The statement is read as “**twoDigits** gets the value of **new DecimalFormat ("0.00")**.”



Software Engineering Observation 4.7

Normally, objects are created with operator **new**. One exception to this is a string literal that is contained in quotes, such as **"hello"**. String literals are treated as objects of class **String** and are instantiated automatically.

Averages do not always evaluate to integer values. Often, an average is a value that contains a fractional part, such as 3.333 or 2.7. These values are referred to as *floating-point numbers* and are represented by the data type **double**. The variable **average** is declared to be of type **double** to capture the fractional result of our calculation. However, the result of the calculation **total / gradeCounter** is an integer, because **total** and **gradeCounter** are both integer variables. Dividing two integers results in *integer division*—any fractional part of the calculation is lost (i.e., *truncated*). The fractional part of the calculation is lost before the result can be assigned to **average**, because the calculation is performed before the assignment occurs.

To perform a floating-point calculation with integer values, we must create temporary values that are floating-point numbers for the calculation. Java provides the *unary cast operator* to accomplish this task. Line 53,

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

uses the cast operator **(double)** to create a temporary floating-point copy of its operand—**total**. Using a cast operator in this manner is called *explicit conversion*. The value stored in **total** is still an integer. The calculation now consists of a floating-point value (the temporary **double** version of **total**) divided by the integer **gradeCounter**. Java knows how to evaluate only arithmetic expressions in which the operands’ data types are identical. To ensure that the operands are of the same type, Java performs an operation called *promotion* (or *implicit conversion*) on selected operands. For example, in an expression containing the data types **int** and **double**, the values of **int** operands are *promoted* to **double** values for use in the expression. In this example, Java promotes the value of **gradeCounter** to type **double**, and then the program performs the calculation and assigns the result of the floating-point division to **average**. Later in this chapter, we discuss all of the standard data types and their order of promotion.



Common Programming Error 4.9

The cast operator can be used to convert between primitive numeric types and to convert between related class types (as we discuss in Chapter 9). Casting a variable to the wrong type may cause compilation errors or runtime errors.

Cast operators are available for any data type. The cast operator is formed by placing parentheses around the name of a data type. The operator is a *unary operator* (i.e., an operator that takes only one operand). In Chapter 2, we studied the binary arithmetic operators. Java also supports unary versions of the plus (+) and minus (−) operators, so the pro-

grammer 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 `%` and one level lower than that of parentheses. (See the operator precedence chart in Appendix C.) 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.



Common Programming Error 4.10

Using floating-point numbers in a manner that assumes they are represented precisely can lead to incorrect results. Floating-point numbers are represented approximately by computers.



Common Programming Error 4.11

Assuming that integer division rounds (rather than truncates) can lead to incorrect results.



Good Programming Practice 4.11

Do not compare floating-point values for equality or inequality. Rather, test for whether the absolute value of the difference between two floating-point numbers is less than a specified small value.

Despite the fact that floating-point numbers are not always 100% precise, they have numerous applications. For example, when we speak of a “normal” body temperature of 98.6, we do not need to be precise to a large number of digits. When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643. The point here is that calling this number simply 98.6 is fine for most applications.

Another way in which floating-point numbers develop is through division. When we divide 10 by 3, the result is 3.333333..., 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.

4.10 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 3 (Nested Control Structures)

Let us work through another complete problem. We once again formulate the algorithm using pseudocode and top-down, stepwise refinement, and we develop a corresponding Java program. Consider the following problem statement:

A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, several of the students who completed this course took the licensing examination. Naturally, the college wants to know how well its students did on the exam. You have been asked to write a program to summarize the results. You have been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam and 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 of students who failed.*
- 4. If more than 8 students passed the exam, print the message “Raise tuition.”*

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

1. The program must process test results for 10 students. A counter-controlled loop will be used.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine if the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it is a 2. (An exercise at the end of the chapter 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 of students who failed the exam.
4. After the program has processed all the results, it must decide if more than eight students passed the exam.

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

Analyze exam results and decide if tuition should be raised

Once again, it is important to emphasize that the top is a complete representation of the program, but several refinements are likely before the pseudocode can evolve naturally into a Java program. Our first refinement is

Initialize variables

Input the ten exam grades and count passes and failures

Print a summary of the exam results and decide if tuition should be raised

Here, too, even though we have a complete representation of the entire program, further refinement is necessary. We now commit to specific variables. We need counters to record the passes and failures, a counter to control the looping process and a variable to store the user input. The pseudocode statement

Initialize variables

may be refined as follows:

Initialize passes to zero

Initialize failures to zero

Initialize student to one

Notice that only the counters for the number of passes, number of failures and number of students are initialized. The pseudocode statement

Input the ten quiz grades and count passes and failures

requires a loop that successively inputs the result of each exam. Here it is known in advance that there are precisely ten exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., *nested* within the loop) a double-selection structure determines whether each exam result is a pass or a failure and increments the appropriate counter accordingly. The refinement of the preceding pseudocode statement is:

```

While student counter is less than or equal to ten
    Input the next exam result

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

    Add one to student counter

```

Notice the use of blank lines to set off the *if/else* control structure to improve program readability. The pseudocode statement

```

Print a summary of the exam results and decide if tuition should be raised

```

may be refined as follows:

```

Print the number of passes
Print the number of failures

If more than eight students passed
    Print "Raise tuition"

```

The complete second refinement appears in Fig. 4.10. Notice that the pseudocode also uses blank lines to set off the *while* structure for program readability.

```

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

While student counter is less than or equal to ten
    Input the next exam result

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

    Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
    Print "Raise tuition"

```

Fig. 4.10 Pseudocode for examination-results problem.

This pseudocode is now sufficiently refined for conversion to Java. The Java program and two sample executions are shown in Fig. 4.11.

```

1  // Fig. 4.11: Analysis.java
2  // Analysis of examination results.
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class Analysis {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         // initializing variables in declarations
13         int passes = 0,           // number of passes
14             failures = 0,         // number of failures
15             student = 1,         // student counter
16             result;              // one exam result
17         String input,            // user-entered value
18             output;             // output string
19
20         // process 10 students; counter-controlled loop
21         while ( student <= 10 ) {
22
23             // obtain result from user
24             input = JOptionPane.showInputDialog(
25                 "Enter result (1=pass,2=fail)" );
26
27             // convert result to int
28             result = Integer.parseInt( input );
29
30             // process result
31             if ( result == 1 )
32                 passes = passes + 1;
33             else
34                 failures = failures + 1;
35
36             student = student + 1;
37         }
38
39         // termination phase
40         output = "Passed: " + passes +
41             "\nFailed: " + failures;
42
43         if ( passes > 8 )
44             output = output + "\nRaise Tuition";
45
46         JOptionPane.showMessageDialog( null, output,
47             "Analysis of Examination Results",
48             JOptionPane.INFORMATION_MESSAGE );
49
50         System.exit( 0 ); // terminate application

```

Fig. 4.11 Java program for examination-results problem (part 1 of 2).

```

51
52     } // end method main
53
54 } // end class Analysis

```

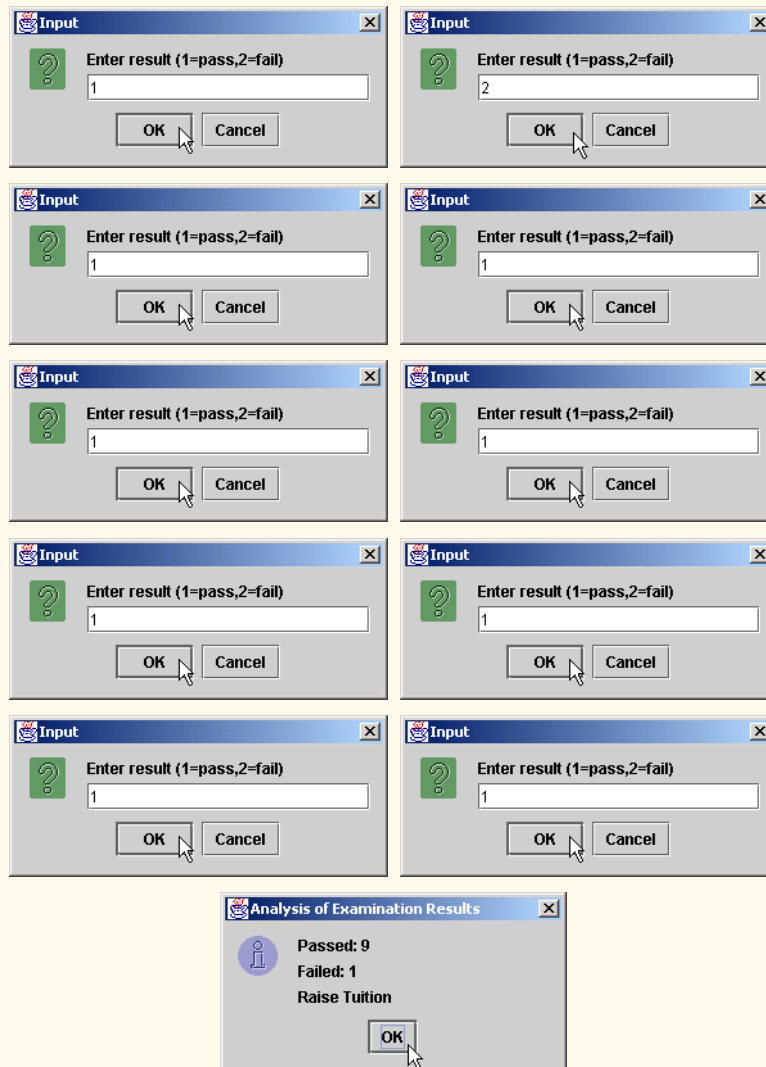


Fig. 4.11 Java program for examination-results problem (part 2 of 2).

Lines 13–18,

```

int passes = 0,           // number of passes
    failures = 0,         // number of failures
    student = 1,          // student counter
    result;               // one exam result
String input,             // user-entered value
    output;               // output string

```

declare the variables used in **main** to process the examination results. Note that we have taken advantage of a feature of Java that incorporates variable initialization into declarations (**passes** is assigned **0**, **failures** is assigned **0** and **student** is assigned **1**). Looping programs may require initialization at the beginning of each repetition; such initialization would normally occur in assignment statements.

Notice the nested **if/else** structure at lines 31–34 of the **while** structure’s body. Also, notice the use of **String** reference **output** in lines 40, 41 and 44 to build the string that lines 46–48 display in a message dialog.



Good Programming Practice 4.12

Initializing local variables when they are declared in methods helps the programmer avoid compiler messages warning of uninitialized data.



Software Engineering Observation 4.8

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, the process of producing a working Java program from the algorithm is normally straightforward.



Software Engineering Observation 4.9

Many experienced programmers write programs without ever using program development tools like pseudocode. These programmers 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 method may work for simple and familiar problems, it can lead to serious errors in large, complex projects.

4.11 Assignment Operators

Java provides several assignment operators for abbreviating assignment expressions. For example, you can abbreviate the statement

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

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

```
c += 3;
```

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. Any statement of the form

```
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;
```

Thus, the assignment expression **c += 3** adds **3** to **c**. Figure 4.12 shows the arithmetic assignment operators, sample expressions using the operators and explanations of what the operators do.

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

Fig. 4.12 Arithmetic assignment operators.

**Performance Tip 4.1**

Programmers can write programs a bit faster and compilers can compile programs a bit faster when the abbreviated assignment operators are used. Some compilers generate code that runs faster when abbreviated assignment operators are used.

**Performance Tip 4.2**

Many of the performance tips we mention in this text result in nominal improvements, so the reader may be tempted to ignore them. Significant performance improvement is often realized when a supposedly nominal improvement is placed in a loop that may repeat a large number of times.

4.12 Increment and Decrement Operators

Java provides the unary *increment operator*, `++`, and the unary *decrement operator*, `--`, which are summarized in Fig. 4.13. A program can increment the value of a variable called `c` by 1 using the increment operator, `++`, rather than the expression `c = c + 1` or `c += 1`. If an increment or decrement operator is placed before a variable, it is referred to as the *pre-increment* or *predecrement operator*, respectively. If an increment or decrement operator is placed after a variable, it is referred to as the *postincrement* or *postdecrement operator*, respectively.

Operator	Called	Sample expression	Explanation
<code>++</code>	preincrement	<code>++a</code>	Increment <code>a</code> by 1, then use the new value of <code>a</code> in the expression in which <code>a</code> resides.
<code>++</code>	postincrement	<code>a++</code>	Use the current value of <code>a</code> in the expression in which <code>a</code> resides, then increment <code>a</code> by 1.
<code>--</code>	predecrement	<code>--b</code>	Decrement <code>b</code> by 1, then use the new value of <code>b</code> in the expression in which <code>b</code> resides.
<code>--</code>	postdecrement	<code>b--</code>	Use the current value of <code>b</code> in the expression in which <code>b</code> resides, then decrement <code>b</code> by 1.

Fig. 4.13 The increment and decrement operators.

Preincrementing (predecrementing) a variable causes the variable to be incremented (decremented) by 1, and then the new value of the variable is used in the expression in which it appears. Postincrementing (postdecrementing) the variable causes the current value of the variable to be used in the expression in which it appears, and then the variable value is incremented (decremented) by 1.

The application in Fig. 4.14 demonstrates the difference between the preincrementing version and the postincrementing version of the `++` increment operator. Postincrementing the variable `c` causes it to be incremented after it is used in the `System.out.println` method call (line 13). Preincrementing the variable `c` causes it to be incremented before it is used in the `System.out.println` method call (line 20).

The program displays the value of `c` before and after the `++` operator is used. The decrement operator (`--`) works similarly.



Good Programming Practice 4.13

Unary operators should be placed next to their operands, with no intervening spaces.

```

1  // Fig. 4.14: Increment.java
2  // Preincrementing and postincrementing
3
4  public class Increment {
5
6      // main method begins execution of Java application
7      public static void main( String args[] )
8      {
9          int c;
10
11          c = 5;
12          System.out.println( c );    // print 5
13          System.out.println( c++ ); // print 5 then postincrement
14          System.out.println( c );    // print 6
15
16          System.out.println();        // skip a line
17
18          c = 5;
19          System.out.println( c );    // print 5
20          System.out.println( ++c ); // preincrement then print 6
21          System.out.println( c );    // print 6
22
23      } // end method main
24
25  } // end class Increment

```

```

5
5
6

5
6
6

```

Fig. 4.14 The difference between preincrementing and postincrementing.

Line 16,

```
System.out.println();           // skip a line
```

uses **System.out.println** to output a blank line. If **println** receives no arguments, it simply outputs a newline character.

The arithmetic assignment operators and the increment and decrement operators can be used to simplify program statements. For example, the three assignment statements in Fig. 4.11 (lines 32, 34 and 36),

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

can be written more concisely with assignment operators as

```
passes += 1;
failures += 1;
student += 1;
```

with preincrement operators as

```
++passes;
++failures;
++student;
```

or with postincrement operators as

```
passes++;
failures++;
student++;
```

It is important to note here that when incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the same effect, and the predecrement and postdecrement forms have the same effect. It is only when a variable appears in the context of a larger expression that preincrementing and post-incrementing the variable have different effects (and similarly for predecrementing and postdecrementing).



Common Programming Error 4.12

*Attempting to use the increment or decrement operator on an expression other than an lvalue is a syntax error. An lvalue is a variable or expression that can appear on the left side of an assignment operation. For example, writing **++(x + 1)** is a syntax error, because **(x + 1)** is not an lvalue.*

The chart in Fig. 4.15 shows the precedence and associativity of the operators that have been introduced up to this point. The operators are shown from top to bottom in decreasing order of precedence. The second column describes the associativity of the operators at each level of precedence. Notice that the conditional operator (**?:**), the unary operators increment (**++**), decrement (**--**), plus (**+**), minus (**-**) and casts and the assignment operators **=**, **+=**, **-=**, ***=**, **/=** and **%=** associate from right to left. All other operators in the operator precedence chart in Fig. 4.15 associate from left to right. The third column names the groups of operators.

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

Fig. 4.15 Precedence and associativity of the operators discussed so far.

4.13 Primitive Data Types

The table in Fig. 4.16 lists the primitive data types in Java. The primitive types are the building blocks for more complicated types. Like its predecessor languages C and C++, Java requires all variables to have a type before they can be used in a program. For this reason, Java is referred to as a *strongly typed language*.

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



Portability Tip 4.1

Unlike in the programming languages C and C++, the primitive types in Java are portable across all computer platforms that support Java. This and many other portability features of Java enable programmers to write programs once, without knowing which computer platform will execute the program. This attribute is sometimes referred to as WORA (Write Once Run Anywhere).

Each data type in Fig. 4.16 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 it to be maximally portable, they chose to use internationally recognized standards for both character formats (Unicode) and floating-point numbers (IEEE 754).

When instance variables of the primitive data types are declared in a class, they are automatically assigned default values unless specified otherwise by the programmer. Instance variables of types **char**, **byte**, **short**, **int**, **long**, **float** and **double** are all given the value **0** by default. Variables of type **boolean** are given the value **false** by default.

Type	Size in bits	Values	Standard
boolean	8	true or false	
char	16	'\u0000' to '\uFFFF' (0 to 65535)	(ISO Unicode character set)
byte	8	−128 to +127 (-2^7 to $2^7 - 1$)	
short	16	−32,768 to +32,767 (-2^{15} to $2^{15} - 1$)	
int	32	−2,147,483,648 to +2,147,483,647 (-2^{31} to $2^{31} - 1$)	
long	64	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (-2^{63} to $2^{63} - 1$)	
float	32	Negative range: −3.4028234663852886E+38 to −1.40129846432481707e−45 Positive range: 1.40129846432481707e−45 to 3.4028234663852886E+38	(IEEE 754 floating point)
double	64	Negative range: −1.7976931348623157E+308 to −4.94065645841246544e−324 Positive range: 4.94065645841246544e−324 to 1.7976931348623157E+308	(IEEE 754 floating point)

Fig. 4.16 The Java primitive data types.

4.14 (Optional Case Study) Thinking About Objects: Identifying Class Attributes

In “Thinking About Objects,” Section 3.8, we began the first phase of an object-oriented design (OOD) for our elevator simulator—identifying the classes needed to implement the simulator. We began by listing the nouns in the problem statement and then created a separate class for each category of noun and noun phrase that perform an important duty in the elevator simulation. We then represented the classes and their relationships in a UML class diagram (Fig. 3.23). Classes have *attributes* (data) and *operations* (behaviors). Class attributes are implemented in Java programs as variables; class behaviors are implemented as methods. In this section, we determine many of the class attributes needed to implement the elevator simulator. In Chapter 5, we examine how these attributes represent an object’s *state*, or condition. In Chapter 6, we determine class behavior. In Chapter 7, we concentrate on the interactions, often called *collaborations*, between the objects in the elevator simulator.

Consider the attributes of some real-world objects: A person's attributes include height and weight, for example. A radio's attributes include its station setting, its volume setting and whether it is set to AM or FM. A car's attributes include its speedometer and odometer readings, the amount of gas in its tank, what gear it is in, etc. A personal computer's attributes include its manufacturer (e.g., Sun, Apple, IBM or Compaq), type of screen (e.g., monochrome or color), main memory size (in megabytes), hard disk size (in gigabytes), etc.

We can identify the attributes of the classes in our system by looking for descriptive words and phrases in the problem statement. For each descriptive word or phrase we find, we create an attribute and assign that attribute to a class. We also create attributes to represent any additional data that a class may need (as the need for this data becomes clear throughout the design process).

We begin examining the problem statement looking for attributes distinct to each class. Figure 4.17 lists the words or phrases from the problem statement that describe each class. The sentence "The user can create any number of people in the simulation" implies that the model will introduce several **Person** objects during execution. We require an integer value representing the number of people in the simulation at any given time, because we may wish to track, or identify, the people in our model. As mentioned in Section 2.9, the **ElevatorModel** object acts as the "representative" for the model (even though the model consists of several classes) for interactions with other parts of the system (in this case, the user is a part of the system), so we assign the **numberOfPeople** attribute to class **ElevatorModel**.

Class **Elevator** contains several attributes. The phrases "is moving" and "is summoned" describe possible states of **Elevator** (we introduce states in the next "Thinking About Objects" section), so we include **moving** and **summoned** as **boolean** attributes. **Elevator** also arrives at a "destination floor," so we include the attribute **destinationFloor**, representing the **Floor** at which the **Elevator** will arrive. Although the problem statement does not mention explicitly that the **Elevator** leaves from a current **Floor**, we may assume another attribute called **currentFloor** representing on which **Floor** the **Elevator** is resting. The problem statement specifies that "both the elevator and each floor have capacity for only one person," so we include the capacity attribute for class **Elevator** (and class **Floor**) and set the value to **1**. Lastly, the problem statement specifies that the elevator "takes five seconds to travel between floors," so we introduce the **travelTime** attribute and set the value to **5**.

Class **Person** contains several attributes. The user must be able to "create a unique person," which implies that each **Person** object should have a unique identifier. We assign integer attribute **ID** to the **Person** object. The **ID** attribute helps to identify that **Person** object. In addition, the problem statement specifies that the **Person** can be "waiting on that floor to enter the elevator." Therefore, "waiting" is a state that **Person** object may enter. Though not mentioned explicitly, if the **Person** is not waiting for the **Elevator**, the **Person** is moving to (or away from) the **Elevator**. We assign the **boolean** attribute **moving** to class **Person**. When this attribute is set to **false**, the **Person** is "waiting." Lastly, the phrase "on that floor" implies that the **Person** occupies a floor. We cannot assign a **Floor** reference to class **Person**, because we are interested only in attributes. However, we want to include the location of the **Person** object in the model, so we include the **currentFloor** attribute, which may have a value of either **1** or **2**.

Class **Floor** has a **capacity** attribute. The problem statement specified that the user could situate the person on either “the first or second floor”—therefore, a **Floor** object requires a value that distinguishes that **Floor** object as the first or second floor, so we include the **floorNumber** attribute.

According to the problem statement, the **ElevatorButton** and **FloorButton** are “pressed” by a **Person**. The buttons may be “reset” as well. The state of each button is either “pressed” or “reset.” We include the **boolean** attribute **pressed** in both button classes. When **pressed** is **true**, the button object is pressed; when **pressed** is **false**, the button object is reset. Classes **ElevatorDoor** and **FloorDoor** exhibit similar characteristics. Both objects are either “open” or “closed,” so we include the **boolean** attribute **open** in both door classes. Class **Light** also falls into this category—the light is either “illuminated” (turned on) or “turned off,” so we include the **boolean** attribute **on** in class **Light**. Note that although the problem statement mentions that the bell rings, there is no mention of when the bell “is ringing,” so we do not include a separate **ring** attribute for class **Bell**. As we progress through this case study, we will continue to add, modify and delete information about the classes in our system.

Class	Descriptive words and phrases
ElevatorModel	number of people in the simulation
ElevatorShaft	[no descriptive words or phrases]
Elevator	moving summoned current floor destination floor capacity of only one person five seconds to travel between floors
Person	unique waiting / moving current floor
Floor	first or second; capacity for only one person
FloorButton	pressed / reset
ElevatorButton	pressed / reset
FloorDoor	door closed / door open
ElevatorDoor	door closed / door open
Bell	[no descriptive words or phrases]
Light	illuminated / turned off

Fig. 4.17 Descriptive words and phrases from problem statement.

Figure 4.18 is a class diagram that lists some of the attributes for each class in our system—the descriptive words and phrases in Fig. 4.17 help us generate these attributes. Note that Fig. 4.18 does not show associations among objects—we showed these associations in Fig. 3.23. In the UML class diagram, a class’s attributes are placed in the middle compartment of the class’s rectangle. Consider the **open** attribute of class **ElevatorDoor**:

```
open : Boolean = false
```

This listing contains three pieces of information about the attribute. The *attribute name* is **open**. The *attribute type* is **Boolean**.² The type depends on the language used to write the software system. In Java, for example, the value can be a *primitive type*, such as **boolean** or **float**, as well as a user-defined type like a class—we begin our study of classes in Chapter 8, where we will see that each new class is a new data type.

We can also indicate an initial value for each attribute. The **open** attribute in class **ElevatorDoor** has an initial value of **false**. If a particular attribute has no initial value specified, only its name and type (separated by a colon) are shown. For example, the **ID** attribute of class **Person** is an integer—in Java, the **ID** attribute is of type **int**. Here we show no initial value, because the value of this attribute is a number that we do not yet know; this number will be determined by **ElevatorModel** at execution time. Integer attribute **currentFloor** for class **Person** is not determined until program execution as well—this attribute is determined when the simulation user decides on which **Floor** to place the **Person**. For now we do not concern ourselves with the types or initial values of the attributes. We include only the information we can glean easily from the problem statement.

Note that Fig. 4.18 does not include attributes for class **ElevatorShaft**. Actually, class **ElevatorShaft** contains seven attributes that we can determine from the class diagram of Fig. 3.23—references to the **Elevator** object, two **FloorButton** objects, two **FloorDoor** objects and two **Light** objects. Class **ElevatorModel** contains three user-defined attributes—two references to **Floor** objects and a reference to the **ElevatorShaft** object. Class **Elevator** also contains three user-defined attributes—reference to the **ElevatorButton** object, the **ElevatorDoor** object and the **Bell** object. To save space, we will not show these additional attributes in our class diagrams—we will, however, include them in the code in the appendices.

The class diagram of Fig. 4.18 provides a good basis for the structure of our model but the diagram is not fully complete. For example, the attribute **currentFloor** in class **Person** represents the floor on which a person is currently located. However, on what floor is the person when that person rides the elevator? These attributes do not yet sufficiently represent the structure of the model. As we present more of the UML and object-oriented design through Chapter 22, we will continue to strengthen the structure of our model.

2. Note that the attribute types in Fig. 4.18 are in UML notation. We will associate the attribute types **Boolean** and **Integer** in the UML diagram with the attribute types **boolean** and **int** in Java, respectively. We described in Chapter 3 that Java provides a “type-wrapper class” for each primitive data type. The Java type-wrapper classes have the same notation as the UML notation for attribute types; however, when we implement our design in Java starting in Chapter 8, we use primitive data types for simplification. Deciding whether to use primitive data types or type-wrapper classes is an implementation-specific issue that should not be mentioned in the UML.

ElevatorModel	ElevatorShaft	ElevatorDoor
numberOfPeople : Integer = 0	<none yet>	open : Boolean = false
Person	Floor	Light
ID : Integer moving : Boolean = true currentFloor : Integer	floorNumber : Integer capacity : Integer = 1	lightOn : Boolean = false
Elevator	ElevatorButton	Bell
moving : Boolean = false summoned : Boolean = false currentFloor : Integer = 1 destinationFloor : Integer = 2 capacity : Integer = 1 travelTime : Integer = 5	pressed : Boolean = false	<none yet>
	FloorButton	FloorDoor
	pressed : Boolean = false	open : Boolean = false

Fig. 4.18 Classes with attributes.

SUMMARY

- A procedure for solving a problem in terms of the actions to be executed and the order in which the actions should be executed is called an algorithm.
- Specifying the order in which statements execute in a computer program is called program control.
- Pseudocode helps the programmer “think out” a program before attempting to write it in a programming language, such as Java.
- Top-down, stepwise refinement is a process for refining pseudocode by maintaining a complete representation of the program during each refinement.
- Declarations are messages to the compiler telling it the names and attributes of variables and telling it to reserve space for variables.
- A selection structure chooses among alternative courses of action.
- The **if** selection structure executes an indicated action only when the condition is true.
- The **if/else** selection structure specifies separate actions to execute when the condition is true and when the condition is false.
- When more than one statement should execute where normally only a single statement appears, the statements must be enclosed in braces, forming a block. A block can be placed anywhere a single statement can be placed.
- An empty statement, indicating that no action is to be taken, is indicated by placing a semicolon (;) where a statement would normally be.
- A repetition structure specifies that an action is to be repeated while some condition remains true.
- The format for the **while** repetition structure is

```
while ( condition )
    statement
```

- A value that contains a fractional part is referred to as a floating-point number and is represented by the data type **float** or **double**.
- Unary cast operator (**double**) creates a temporary floating-point copy of its operand.
- Java provides the arithmetic assignment operators **+=**, **-=**, ***=**, **/=** and **%=**, which help abbreviate certain common types of expressions.
- The increment operator, **++**, and the decrement operator, **--**, increment or decrement a variable by 1, respectively. If the operator is prefixed to the variable, the variable is incremented or decremented by 1 first, and then used in its expression. If the operator is postfix to the variable, the variable is used in its expression, and then incremented or decremented by 1.
- The primitive types (**boolean**, **char**, **byte**, **short**, **int**, **long**, **float** and **double**) are the building blocks for more complicated types in Java.
- Java requires all variables to have a type before they can be used in a program. For this reason, Java is referred to as a strongly typed language.
- Primitive types in Java are portable across all computer platforms that support Java.
- Java uses internationally recognized standards for both character formats (Unicode) and floating-point numbers (IEEE 754).
- Instance variables of types **char**, **byte**, **short**, **int**, **long**, **float** and **double** are all given the value **0** by default. Variables of type **boolean** are given the value **false** by default.

TERMINOLOGY

-- operator	integer division
?: operator	ISO Unicode character set
++ operator	logic error
action	loop counter
action/decision model	loop-continuation condition
algorithm	nested control structures
arithmetic assignment operators:	postdecrement operator
+=, -=, *=, /= and %=	postincrement operator
block	predecrement operator
body of a loop	preincrement operator
cast operator, (type)	promotion
conditional operator (?:)	pseudocode
control structure	repetition
counter-controlled repetition	repetition structures
decision	selection
decrement operator (--)	sentinel value
definite repetition	sequential execution
double	single-entry/single-exit control structures
double-selection structure	single-selection structure
empty statement (;)	stacked control structures
if selection structure	structured programming
if/else selection structure	syntax error
implicit conversion	top-down, stepwise refinement
increment operator (++)	unary operator
indefinite repetition	while repetition structure
infinite loop	white-space characters
initialization	

SELF-REVIEW EXERCISES

- 4.1** Fill in the blanks in each of the following statements:
- All programs can be written in terms of three types of control structures: _____, _____ and _____.
 - The _____ selection structure is used to execute one action when a condition is true and another action when that condition is false.
 - Repeating a set of instructions a specific number of times is called _____ repetition.
 - When it is not known in advance how many times a set of statements will be repeated, a _____ value can be used to terminate the repetition.
- 4.2** Write four different Java statements that each add 1 to integer variable **x**.
- 4.3** Write Java statements to accomplish each of the following tasks:
- Assign the sum of **x** and **y** to **z**, and increment the value of **x** by 1 after the calculation. Use only one statement.
 - Test if the value of the variable **count** is greater than 10. If it is, print **"Count is greater than 10"**.
 - Decrement the variable **x** by 1, and then subtract it from the variable **total**. Use only one statement.
 - Calculate the remainder after **q** is divided by **divisor**, and assign the result to **q**. Write this statement in two different ways.
- 4.4** Write a Java statement to accomplish each of the following tasks:
- Declare variables **sum** and **x** to be of type **int**.
 - Assign **1** to variable **x**.
 - Assign **0** to variable **sum**.
 - Add variable **x** to variable **sum**, and assign the result to variable **sum**.
 - Print **"The sum is: "**, followed by the value of variable **sum**.
- 4.5** Combine the statements that you wrote in Exercise 4.4 into a Java application that calculates and prints the sum of the integers from 1 to 10. Use the **while** structure to loop through the calculation and increment statements. The loop should terminate when the value of **x** becomes 11.
- 4.6** Determine the value of each variable after the calculation is performed. Assume that when each statement begins executing, all variables have the integer value 5.
- product *= x++;**
 - quotient /= ++x;**
- 4.7** Identify and correct the errors in each of the following sets of code:
- ```
while (c <= 5) {
 product *= c;
 ++c;
}
```
  - ```
if ( gender == 1 )
    System.out.println( "Woman" );
else;
    System.out.println( "Man" );
```
- 4.8** What is wrong with the following **while** repetition structure?
- ```
while (z >= 0)
 sum += z;
```

**ANSWERS TO SELF-REVIEW EXERCISES**

- 4.1** a) sequence, selection, repetition. b) **if/else**. c) counter-controlled, or definite. d) Sentinel, signal, flag or dummy.

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

4.3    a) `z = x++ + y;`  
       b) `if ( count > 10 )`  
           `System.out.println( "Count is greater than 10" );`  
       c) `total -= --x;`  
       d) `q %= divisor;`  
           `q = q % divisor;`

4.4    a) `int sum, x;`  
       b) `x = 1;`  
       c) `sum = 0;`  
       d) `sum += x; or sum = sum + x;`  
       e) `System.out.println( "The sum is: " + sum );`

4.5    The program is as follows:

```

1 // Calculate the sum of the integers from 1 to 10
2 public class Calculate {
3 public static void main(String args[])
4 {
5 int sum, x;
6
7 x = 1;
8 sum = 0;
9
10 while (x <= 10) {
11 sum += x;
12 ++x;
13 }
14
15 System.out.println("The sum is: " + sum);
16 }
17 }
```

4.6    a) `product = 25, x = 6`  
       b) `quotient = 0, x = 6`

4.7    a) Error: Missing the closing right brace of the **while** structure's body.  
       Correction: Add a closing right brace after the statement `++c;`.  
       b) Error: Semicolon after **else** results in a logic error. The second output statement will always be executed.  
       Correction: Remove the semicolon after **else**.

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

## EXERCISES

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

- a) `if ( age >= 65 );`  
`System.out.println( "Age greater than or equal to 65" );`  
`else`  
`System.out.println( "Age is less than 65" );`
- b) `int x = 1, total;`  
`while ( x <= 10 ) {`  
`total += x;`  
`++x;`  
`}`
- c) `While ( x <= 100 )`  
`total += x;`  
`++x;`
- d) `while ( y > 0 ) {`  
`System.out.println( y );`  
`++y;`

**4.10** What does the following program print?

```

1 public class Mystery {
2
3 public static void main(String args[])
4 {
5 int y, x = 1, total = 0;
6
7 while (x <= 10) {
8 y = x * x;
9 System.out.println(y);
10 total += y;
11 ++x;
12 }
13
14 System.out.println("Total is " + total);
15 }
16 }
```

**For Exercise 4.11 through Exercise 4.14, perform each of the following steps:**

- a) Read the problem statement.
- b) Formulate the algorithm using pseudocode and top-down, stepwise refinement.
- c) Write a Java program.
- d) Test, debug and execute the Java program.
- e) Process three complete sets of data.

**4.11** Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording 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 tankful. The program should calculate and display the miles per gallon obtained for each tankful and print the combined miles per gallon obtained for all tankfuls up to this point. All averaging calculations should produce floating-point results. Use input dialogs to obtain the data from the user.

**4.12** Develop a Java application that will determine if a department-store customer 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, and
- e) allowed credit limit.

The program should input each of these facts from input dialogs as integers, calculate the new balance ( $= \text{beginning balance} + \text{charges} - \text{credits}$ ), display the new balance and determine if the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the message "Credit limit exceeded."

**4.13** A large company pays its salespeople on a commission basis. The salespeople receive \$200 per week, plus 9% of their gross sales for that week. For example, a salesperson who sells \$5000 worth of merchandise in a week receives \$200 plus 9% of \$5000, or a total of \$650. You have been supplied with a list of 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 is no limit to the number of items that can be sold by a salesperson.

**4.14** Develop a Java application that will determine the gross pay for each of three employees. The company pays "straight time" for the first 40 hours worked by each employee and pays "time and a half" for all hours worked in excess of 40 hours. You are given a list of the employees of the company, the number of hours each employee worked last week and the hourly rate of each employee. Your program should input this information for each employee and should determine and display the employee's gross pay. Use input dialogs to input the data.

**4.15** The process of finding the largest value (i.e., the maximum of a group of values) 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 and then a Java application that inputs a series of 10 single-digit numbers as characters and determines and prints the largest of the numbers. *Hint:* Your program should use 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 current digit input to the program;
- c) **largest**: The largest number found so far.

**4.16** Write a Java application that uses looping to print the following table of values:

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

**4.17** Using an approach similar to that for Exercise 4.15, find the *two* largest values of the 10 digits entered. [Note: You may input each number only once.]

**4.18** Modify the program in Fig. 4.11 to validate its inputs. For any input, if the value entered is other than 1 or 2, keep looping until the user enters a correct value.

**4.19** What does the following program print?

```

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

```

**4.20** What does the following program print?

```

1 public class Mystery3 {
2
3 public static void main(String args[])
4 {
5 int row = 10, column;
6
7 while (row >= 1) {
8 column = 1;
9
10 while (column <= 10) {
11 System.out.print(row % 2 == 1 ? "<" : ">");
12 ++column;
13 }
14
15 --row;
16 System.out.println();
17 }
18 }
19 }

```

**4.21** (*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. Note that the compiler ignores the indentation in a Java program. Also, the Java compiler always associates an **else** with the previous **if** unless told to do otherwise by the placement of braces (**{}**). On first glance, the programmer may not be sure which **if** an **else** matches; this situation is referred to as the “dangling-else problem.” We have eliminated the indentation from the following code to make the problem more challenging. [*Hint*: Apply indentation conventions you have learned.]

```

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

```

```

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

```

**4.22** (*Another Dangling-Else Problem*) Modify the given code to produce the output shown in each part of the problem. Use proper indentation techniques. You may not make any changes other than inserting braces and changing the indentation of the code. The compiler ignores indentation in a Java program. We have eliminated the indentation from the given code to make the problem more challenging. [Note: It is possible that no modification is necessary for some of the parts.]

```

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

```

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

```

#####
$$$$$
#####

```

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

```

#####

```

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

```

#####
#####

```

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

```

#####
$$$$$
#####

```

**4.23** Write an applet that reads in the size of the side of a square and displays a hollow square of that size out of asterisks, by using the **drawString** method inside your applet's **paint** method.

Use an input dialog to read the size from the user. Your program should work for squares of all lengths of side between 1 and 20.

**4.24** A palindrome is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers are palindromes: 12321, 55555, 45554 and 11611. Write an application that reads in a five-digit integer and determines whether or not it is a palindrome. If the number is not five digits long, display an error message dialog indicating the problem to the user. When the user dismisses the error dialog, allow the user to enter a new value.

**4.25** 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 modulus and division operators to pick off the “binary number’s” digits one at a time, from right to left. Just as in the decimal number system, where the rightmost digit has a positional value of 1 and the next digit to the left has a positional value of 10, then 100, then 1000, etc., in the binary number system the rightmost digit has a positional value of 1, the next digit to the left has a positional value of 2, then 4, then 8, etc. Thus, the decimal number 234 can be interpreted as  $4 * 1 + 3 * 10 + 2 * 100$ . The decimal equivalent of binary 1101 is  $1 * 1 + 0 * 2 + 1 * 4 + 1 * 8$ , or  $1 + 0 + 4 + 8$  or, 13.]

**4.26** Write an application that displays the following checkerboard pattern:

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

Your program may use only three output statements, one of the form

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

one of the form

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

and one of the form

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

Note that the preceding statement indicates that the program should output a single newline character to drop to the next line of the output. [*Hint:* Repetition structures are required in this exercise.]

**4.27** Write an application that keeps displaying in the command window the multiples of the integer 2, namely 2, 4, 8, 16, 32, 64, etc. Your loop should not terminate (i.e., you should create an infinite loop). What happens when you run this program?

**4.28** What is wrong with the following statement? Provide the correct statement to add one to the sum of **x** and **y**.

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

**4.29** Write an application that reads three nonzero values entered by the user in input dialogs and determines and prints if they could represent the sides of a triangle.

**4.30** Write an application that reads three nonzero integers and determines and prints if they could represent the sides of a right triangle.

**4.31** A company wants to transmit data over the telephone, but it is concerned that its phones may be tapped. All of its data are transmitted as four-digit integers. It has asked you to write a program that will encrypt its data so that the data may be transmitted more securely. Your application should read a four-digit integer entered by the user in an input dialog and encrypt it as follows: Replace each digit by *(the sum of that digit plus 7) modulus 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 to form the original number.

**4.32** The factorial of a nonnegative integer  $n$  is written as  $n!$  (pronounced “ $n$  factorial”) and is defined as follows:

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

and

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

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

- a) Write an application that reads a nonnegative integer from an input dialog and computes and prints its factorial.
- b) Write an application that estimates the value of the mathematical constant  $e$  by using the formula

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

- c) Write an application that computes the value of  $e^x$  by using the formula:

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