Control Statements: Part I

OBJECTIVES

In this chapter you will learn:

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

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

The wheel is come full circle.
—William Shakespeare

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

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

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

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

4.2 Algorithms

Any computing problem can be solved by executing a series of actions in a specific order. A procedure for solving a problem in terms of
1. the actions to execute and
2. the order in which these actions execute

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

Consider the “rise-and-shine algorithm” followed by one executive for getting out of bed and going to work: (1) Get out of bed; (2) take off pajamas; (3) take a shower; (4) get dressed; (5) eat breakfast; (6) carpool to work. This routine gets the executive to work well prepared to make critical decisions. Suppose that the same steps are performed in a slightly different order: (1) Get out of bed; (2) take off pajamas; (3) get dressed; (4) take a shower; (5) eat breakfast; (6) carpool to work. In this case, our executive shows up for work soaking wet.

Specifying the order in which statements (actions) execute in a program is called program control. This chapter investigates program control using Java’s control statements.

### 4.3 Pseudocode

Pseudocode is an informal language that helps programmers develop algorithms without having to worry about the strict details of Java language syntax. The pseudocode we present is particularly useful for developing algorithms that will be converted to structured portions of Java programs. Pseudocode is similar to everyday English—it is convenient and user friendly, but it is not an actual computer programming language. We begin using pseudocode in Section 4.5, and a sample pseudocode program appears in Fig. 4.5.

Pseudocode does not execute on computers. Rather, it helps the programmer “think out” a program before attempting to write it in a programming language, such as Java. This chapter provides several examples of how to use pseudocode to develop Java programs.

The style of pseudocode we present consists purely of characters, so programmers can type pseudocode conveniently, using any text-editor program. A carefully prepared pseudocode program can easily be converted to a corresponding Java program. In many cases, this simply requires replacing pseudocode statements with Java equivalents.

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

### 4.4 Control Structures

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

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software development groups. The blame was pointed at the goto statement (used in most programming languages of the time), which allows 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.” [Note: Java does not have a goto
statement; however, the word goto is reserved by Java and should not be used as an identifier in programs.]

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

Bohm and Jacopini’s work demonstrated that all programs could be written in terms of only three control structures—the sequence structure, the selection structure and the repetition structure. The term “control structures” comes from the field of computer science. When we introduce Java’s implementations of control structures, we will refer to them in the terminology of the Java Language Specification as “control statements.”

**Sequence Structure in Java**

The sequence structure is built into Java. Unless directed otherwise, the computer executes Java statements one after the other in the order in which they are written—that is, in sequence. The activity diagram in Fig. 4.1 illustrates a typical sequence structure in which two calculations are performed in order. Java lets us 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 place several actions in sequence.

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

Fig. 4.1 | Sequence structure activity diagram.

---

4.4 Control Structures

Like pseudocode, activity diagrams help programmers develop and represent algorithms, although many programmers prefer pseudocode. Activity diagrams clearly show how control structures operate.

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

The solid circle located at the top of the activity diagram represents the activity’s initial state—the beginning of the workflow before the program performs the modeled actions. The solid circle surrounded by a hollow circle that appears at the bottom of the diagram represents the final state—the end of the workflow after the program performs its actions.

Figure 4.1 also includes rectangles with the upper-right corners folded over. These are UML notes (like comments in Java)—explanatory remarks that describe the purpose of symbols in the diagram. Figure 4.1 uses UML notes to show the Java code associated with each action state in the activity diagram. A dotted line connects each note with the element that the note describes. Activity diagrams normally do not show the Java code that implements the activity. We use notes for this purpose here to illustrate how the diagram relates to Java code. For more information on the UML, see our optional case study, which appears in the Software Engineering Case Study sections at the ends of Chapters 1–8 and 10, or visit www.uml.org.

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

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

Repetition Statements in Java
Java provides three repetition statements (also called looping statements) that enable programs to perform statements repeatedly as long as a condition (called the loop-continuation condition) remains true. The repetition statements are the while, do...while and for statements. (Chapter 5 presents the do...while and for statements.) The while and for statements perform the action (or group of actions) in their bodies zero or more times—if the loop-continuation condition is initially false, the action (or group of actions) will not execute. The do...while statement performs the action (or group of actions) in its body one or more times.
The words if, else, switch, while, do and for are Java keywords. Recall that keywords are used to implement various Java features, such as control statements. Keywords cannot be used as identifiers, such as variable names. A complete list of Java keywords appears in Appendix C.

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

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

\[
\text{If student's grade is greater than or equal to 60}
\text{Print "Passed"}
\]

determines whether the condition “student’s grade is greater than or equal to 60” is true or false. If it is true, “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. The indentation of the second line of this selection statement is optional, but recommended, because it emphasizes the inherent structure of structured programs.

The preceding pseudocode If statement may be written in Java as

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

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

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

The if statement is a single-entry/single-exit control statement. We will see that the activity diagrams for the remaining control statements also contain initial states, transition arrows, action states that indicate actions to perform, decision symbols (with associated guard conditions) that indicate decisions to be made and final states. This is consistent with the action/decision model of programming we have been emphasizing.

Envision seven bins, each containing only one type of Java control statement. The control statements are all empty. Your task is to assemble a program from as many of each type of control statement as the algorithm demands, combining them in only two possible ways (stacking or nesting), then filling in the action states and decisions with action expressions and guard conditions appropriate for the algorithm. We will discuss the variety of ways in which actions and decisions can be written.

4.6 if...else Double-Selection Statement

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

\[
\text{If student's grade is greater than or equal to 60}
\]
\[
\text{Print "Passed"}
\]
\[
\text{Else}
\]
\[
\text{Print "Failed"}
\]

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

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

```java
if ( grade >= 60 )
    System.out.println( "Passed" );
else
    System.out.println( "Failed" );
```
Note that the body of the else is also indented. Whatever indentation convention you choose should be applied consistently throughout your programs. It is difficult to read programs that do not obey uniform spacing conventions.

**Good Programming Practice 4.1**

Indent both body statements of an `if...else` statement.

**Good Programming Practice 4.2**

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

Figure 4.3 illustrates the flow of control in the `if...else` statement. Once again, the symbols in the UML activity diagram (besides the initial state, transition arrows and final state) represent action states and decisions. We continue to emphasize this action/decision model of computing. Imagine again a deep bin containing as many empty `if...else` statements as might be needed to build any Java program. Your job is to assemble these `if...else` statements (by stacking and nesting) with any other control statements required by the algorithm. You fill in the action states and decision symbols with action expressions and guard conditions appropriate to the algorithm you are developing.

**Conditional Operator (?:)**

Java provides the [conditional operator](https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html#true) `?:` that can be used in place of an `if...else` statement. This is Java’s only [ternary operator](https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html#false) — this means that it takes three operands. Together, the operands and the `?:` symbol form a [conditional expression](https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html#true). The first operand (to the left of the `?`) is a [boolean expression](https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html#false) (i.e., a condition that evaluates to a boolean value — [true](https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html#true) or [false](https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html#false)), the second operand (between the `?` and `:`) is the value of the conditional expression if the boolean expression is [true](https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html#true) and the third operand (to the right of the `:`) is the value of the conditional expression if the boolean expression evaluates to [false](https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html#false). For example, the statement

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

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

**Good Programming Practice 4.3**

Conditional expressions are more difficult to read than if...else statements and should be used to replace only simple if...else statements that choose between two values.

**Nested if...else Statements**

A program can test multiple cases by placing if...else statements inside other if...else statements to create nested if...else statements. For example, the following pseudocode represents a nested if...else that prints A for exam grades greater than or equal to 90, B for grades 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:

```java
if (studentGrade >= 90)
    Print "A"
else
    if (studentGrade >= 80)
        Print "B"
    else
        if (studentGrade >= 70)
            Print "C"
        else
            if (studentGrade >= 60)
                Print "D"
            else
                Print "F"
```

This pseudocode may be written in Java as

```java
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 statement in the if-part of the first if...else statement will execute. After that statement executes, the else-part of the “outermost” if...else statement is skipped. Most Java programmers prefer to write the preceding if...else statement as
The two forms are identical except for the spacing and indentation, which the compiler ignores. The latter form is popular because it avoids deep indentation of the code to the right. Such indentation often leaves little room on a line of code, forcing lines to be split and decreasing program readability.

**Dangling-else Problem**

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

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

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

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

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

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

```java
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 statement is in the body of the first if and that the else is associated with the first if. Exercises 4.27–4.28 investigate the dangling-else problem further.

**Blocks**

The if statement normally expects only one statement in its body. To include several statements in the body of an if (or the body of an else for an if...else statement), enclose the statements in braces ({ and }). A set of statements contained within a pair of braces is called a block. A block can be placed anywhere in a program that a single statement can be placed.

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

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

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

Failed.
You must take this course again.

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

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

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

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

---

**Common Programming Error 4.1**

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

**Good Programming Practice 4.4**

Always using braces in an if...else (or other) statement helps prevent their accidental omission, especially when adding statements to the if-part or the else-part at a later time. To avoid omitting one or both of the braces, some programmers type the beginning and ending braces of blocks before typing the individual statements within the braces.

Just as a block can be placed anywhere a single statement can be placed, it is also possible to have an empty statement. Recall from Section 2.8 that the empty statement is represented by placing a semicolon (;) where a statement would normally be.
Common Programming Error 4.2
Placing a semicolon after the condition in an if or if...else statement leads to a logic error in single-selection if statements and a syntax error in double-selection if...else statements (when the if-part contains an actual body statement).

4.7 while Repetition Statement
A repetition statement (also called a looping statement or a loop) allows the programmer to specify that a program should repeat an action while some condition remains true. The pseudocode statement

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

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

As an example of Java’s while repetition statement, consider a program segment designed to find the first power of 3 larger than 100. Suppose that the int variable product is initialized to 3. When the following while statement finishes executing, product contains the result:

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

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

Common Programming Error 4.3
Not providing, in the body of a while statement, an action that eventually causes the condition in the while to become false normally results in a logic error called an infinite loop, in which the loop never terminates.

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

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

4.8 Formulating Algorithms: Counter-Controlled Repetition

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

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

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

**Pseudocode Algorithm with Counter-Controlled Repetition**

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

**Software Engineering Observation 4.1**

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

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

**Implementing Counter-Controlled Repetition in Class GradeBook**

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

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

Note that the declarations (in lines 42–45) appear in the body of method determineClassAverage. Recall that variables declared in a method body are local variables and can be used only from the line of their declaration in the method to the closing right brace (}).

---

**Fig. 4.5** Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.
4.8 Formulating Algorithms: Counter-Controlled Repetition

// Fig. 4.6: GradeBook.java
// GradeBook class that solves class-average problem using
// counter-controlled repetition.
import java.util.Scanner; // program uses class Scanner

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

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

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

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

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

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

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

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

        // processing phase
        while ( gradeCounter <= 10 ) // loop 10 times
        {
            Fig. 4.6 | Counter-controlled repetition: Class-average problem. (Part 1 of 2.)
of the method declaration. A local variable’s declaration must appear before the variable is used in that method. A local variable cannot be accessed outside the method in which it is declared.

In the versions of class `GradeBook` in this chapter, we simply read and process a set of grades. The averaging calculation is performed in method `determineClassAverage` using local variables—we do not preserve any information about student grades in instance variables of the class. In later versions of the class (in Chapter 7, Arrays), we maintain the grades in memory using an instance variable that refers to a data structure known as an array. This allows a `GradeBook` object to perform various calculations on the same set of grades without requiring the user to enter the grades multiple times.

**Good Programming Practice 4.5**

Separate declarations from other statements in methods with a blank line for readability.

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

**Common Programming Error 4.4**

Reading the value of a local variable before it is initialized results in a compilation error. All local variables must be initialized before their values are read in expressions.

**Error-Prevention Tip 4.1**

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

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

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

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

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

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

Notes on Integer Division and Truncation
The averaging calculation performed by method determineClassAverage in response to the method call at line 14 in Fig. 4.7 produces an integer result. The program's output

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

**Fig. 4.7** GradeBookTest class creates an object of class GradeBook (Fig. 4.6) and invokes its determineClassAverage method. (Part 1 of 2)
Welcome to the grade book for
CS101 Introduction to Java Programming!

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

Total of all 10 grades is 846
Class average is 84

Fig. 4.7 | GradeBookTest class creates an object of class GradeBook (Fig. 4.6) and invokes its
determineClassAverage method. (Part 2 of 2.)

indicates that the sum of the grade values in the sample execution is 846, which, when
dived by 10, should yield the floating-point number 84.6. However, the result of the calculation
tota1 / 10 (line 61 of Fig. 4.6) is the integer 84, because tota1 and 10 are both
egers. Dividing two integers results in integer division—any fractional part of the calcu-
ation is lost (i.e., truncated). We will see how to obtain a floating-point result from the
aving calculation in the next section.

Common Programming Error 4.5

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

4.9 Formulating Algorithms: Sentinel-Controlled
Repetition

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

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

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

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

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

**Common Programming Error 4.6**

Choosing a sentinel value that is also a legitimate data value is a logic error.

**Developing the Pseudocode Algorithm with Top-Down, Stepwise Refinement:**

The Top and First Refinement

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

*Determine the class average for the quiz*

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

*Initialize variables*

*Input, sum and count the quiz grades*

*Calculate and print the class average*

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

**Software Engineering Observation 4.2**

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

**Software Engineering Observation 4.3**

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

**Proceeding to the Second Refinement**

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

*Initialize variables*
can be refined as follows:

- Initialize total to zero
- Initialize counter to zero

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

The pseudocode statement

**Input, sum and count the quiz grades**

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

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

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

The pseudocode statement

**Calculate and print the class average**

can be refined as follows:

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

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

---

**Error-Prevention Tip 4.2**

> When performing division by an expression whose value could be zero, explicitly test for this possibility and handle it appropriately in your program (e.g., by printing an error message) rather than allow the error to occur.
4.9 Formulating Algorithms: Sentinel-Controlled Repetition

Class-average problem pseudocode algorithm with sentinel-controlled repetition:

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

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

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

**Software Engineering Observation 4.4**

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

**Software Engineering Observation 4.5**

Some experienced programmers write programs without using program development tools like pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this may work for simple and familiar problems, it can lead to serious errors and delays in large, complex projects.

**Implementing Sentinel-Controlled Repetition in Class GradeBook**

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

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

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

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

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

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

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

    // determine the average of an arbitrary number of grades
    public void determineClassAverage() {
        Scanner input = new Scanner(System.in);

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

        // initialization phase
        total = 0; // initialize total
        gradeCounter = 0; // initialize loop counter
```
// processing phase
// prompt for input and read grade from user
System.out.print( "Enter grade or -1 to quit: " );
grade = input.nextInt();

// loop until sentinel value read from user
while ( grade != -1 )
{
    total = total + grade; // add grade to total
    gradeCounter = gradeCounter + 1; // increment counter
    // prompt for input and read next grade from user
    System.out.print( "Enter grade or -1 to quit: " );
    grade = input.nextInt();
} // end while

// termination phase
// if user entered at least one grade...
if ( gradeCounter != 0 )
{
    // calculate average of all grades entered
    average = (double) total / gradeCounter;
    // display total and average (with two digits of precision)
    System.out.printf( "\nTotal of the %d grades entered is %.1f\n", gradeCounter, total );
    System.out.printf( "Class average is %.2f\n", average );
} // end if
else // no grades were entered, so output appropriate message
System.out.println( "No grades were entered" );
} // end method determineClassAverage

Fig. 4.9 | Sentinel-controlled repetition: Class-average problem. (Part 2 of 2.)

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

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

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

**Good Programming Practice 4.6**

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

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

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

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

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

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

**Common Programming Error 4.7**

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

**Explicitly and Implicitly Converting Between Primitive Types**

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

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

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

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

### Common Programming Error 4.8

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

Cast operators are available for any type. The cast operator is formed by placing parentheses around the name of a 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 programmer can write expressions like −7 or +5. Cast operators associate from right to left and have the same precedence as other unary operators, such as unary + and unary −. This precedence is one level higher than that of the **multiplicative operators** *, / and %. (See the operator precedence chart in Appendix A.) We indicate the cast operator with the notation (type) in our precedence charts, to indicate that any type name can be used to form a cast operator.

Line 77 outputs the class average using `System.out.printf` method. In this example, we display the class average rounded to the nearest hundredth. The format specifier `.2f` in `printf`’s format control string (line 77) indicates that variable average’s value should be displayed with two digits of precision to the right of the decimal point—indicated by .2 in the format specifier. The three grades entered during the sample execution of class `GradeBookTest` (Fig. 4.10) total 257, which yields the average 85.666666. Method `printf` uses the precision in the format specifier to round the value to the specified

---

```java
1 // Fig. 4.10: GradeBookTest.java
2 // Create GradeBook object and invoke its determineClassAverage method.
3 public class GradeBookTest
4 {
5     public static void main( String args[] )
6     {
7         // Fig. 4.10 | GradeBookTest class creates an object of class GradeBook (Fig. 4.9) and invokes its determineClassAverage method. (Part 1 of 2.)
```

number of digits. In this program, the average is rounded to the hundredths position and the average is displayed as 85.67.

4.10 Formulating Algorithms: Nested Control Statements

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

Consider the following problem statement:

A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You 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 or a 2 if the student failed.

Your program should analyze the results of the exam as follows:

1. Input each test result (i.e., a 1 or a 2). Display the message “Enter result” on the screen each time the program requests another test result.
2. Count the number of test results of each type.
3. Display a summary of the test results indicating the number of students who passed and the number who failed.
4. If more than eight students passed the exam, print the message “Raise tuition.”
After reading the problem statement carefully, we make the following observations:

1. The program must process test results for 10 students. A counter-controlled loop can be used because the number of test results is known in advance.

2. Each test result has a numeric value—either a 1 or a 2. Each time the program reads a test result, the program must determine whether 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. (Exercise 4.24 considers the consequences of this assumption.)

3. Two counters are used to keep track of the exam results—one to count the number of students who passed the exam and one to count the number of students who failed the exam.

4. After the program has processed all the results, it must decide whether 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 whether tuition should be raised*

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

Our first refinement is

*Initialize variables*

*Input the 10 exam results, and count passes and failures*

*Print a summary of the exam results and decide whether 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. Counters are needed to record the passes and failures, a counter will be used to control the looping process and a variable is needed to store the user input. The variable in which the user input will be stored is not initialized at the start of the algorithm, because its value is read from the user during each iteration of the loop.

The pseudocode statement

*Initialize variables*

can be refined as follows:

*Initialize passes to zero*

*Initialize failures to zero*

*Initialize student counter to one*

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

The pseudocode statement

*Input the 10 exam results, and count passes and failures*

requires a loop that successively inputs the result of each exam. We know in advance that there are precisely 10 exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., nested within the loop), a double-selection structure will determine whether each exam result is a pass or a failure and will increment the appropriate counter. The refinement of the preceding pseudocode statement is then
While student counter is less than or equal to 10
  Prompt the user to enter the next exam result
  Input the next exam result
  If the student passed
    Add one to passes
  Else
    Add one to failures
  Add one to student counter

We use blank lines to isolate the If...Else control structure, which improves readability. The pseudocode statement

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

can be refined as follows:

  Print the number of passes
  Print the number of failures
  If more than eight students passed
    Print "Raise tuition"

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

The complete second refinement appears in Fig. 4.11. Notice that blank lines are also used to set off the While structure for program readability. This pseudocode is now sufficiently refined for conversion to Java. The Java class that implements the pseudocode algorithm is shown in Fig. 4.12, and two sample executions appear in Fig. 4.13.

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

**Fig. 4.11** | Pseudocode for examination-results problem.
4.10 Formulating Algorithms: Nested Control Statements

```java
// Fig. 4.12: Analysis.java
// Analysis of examination results.
import java.util.Scanner; // class uses class Scanner

public class Analysis {
    public void processExamResults() {
        // create Scanner to obtain input from command window
        Scanner input = new Scanner( System.in );

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

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

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

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

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

        // determine whether more than 8 students passed
        if ( passes > 8 )
            System.out.println( "Raise Tuition" );
    } // end method processExamResults
}
} // end class Analysis
```

**Fig. 4.12** | Nested control structures: Examination-results problem.

Lines 13–16 of Fig. 4.12 declare the variables that method processExamResults of class Analysis uses to process the examination results. Several of these declarations use Java’s ability to incorporate variable initialization into declarations (passes is assigned 0, failures is assigned 0 and studentCounter is assigned 1). Looping programs may require initialization at the beginning of each repetition—such reinitialization would normally be performed by assignment statements rather than in declarations.
The while statement (lines 19–33) loops 10 times. During each iteration, the loop inputs and processes one exam result. Notice that the if...else statement (lines 26–29) for processing each result is nested in the while statement. If the result is 1, the if...else statement increments passes; otherwise, it assumes the result is 2 and increments failures. Line 32 increments studentCounter before the loop condition is tested again at line 19. After 10 values have been input, the loop terminates and line 36 displays the number of passes and failures. The if statement at lines 39–40 determines whether more than eight students passed the exam and, if so, outputs the message "Raise Tuition".

**Error-Prevention Tip 4.3**

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

**AnalysisTest Class That Demonstrates Class Analysis**

Class AnalysisTest (Fig. 4.13) creates an Analysis object (line 8) and invokes the object’s processExamResults method (line 9) to process a set of exam results entered by the user. Figure 4.13 shows the input and output from two sample executions of the program. During the first sample execution, the condition at line 39 of method processExamResults in Fig. 4.12 is true—more than eight students passed the exam, so the program outputs a message indicating that the tuition should be raised.

```java
1 // Fig. 4.13: AnalysisTest.java
2 // Test program for class Analysis.
3
4 public class AnalysisTest
5 {
6    public static void main(String args[])
7    {
8        Analysis application = new Analysis(); // create Analysis object
9        application.processExamResults(); // call method to process results
10    } // end main
11
12} // end class AnalysisTest
```

Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Raise Tuition

**Fig. 4.13** | Test program for class Analysis (Fig. 4.12). (Part 1 of 2.)
4.11 Compound Assignment Operators

Java provides several compound assignment operators for abbreviating assignment expressions. Any statement of the form

\[ \text{variable} = \text{variable operator expression}; \]

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

\[ \text{variable operator=} \text{expression}; \]

For example, you can abbreviate the statement

\[ c = c + 3; \]

with the addition compound assignment operator, \(+=\), as

\[ c += 3; \]

The \(+=\) operator adds the value of the expression on 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. Thus, the assignment expression \(c += 3\) adds 3 to \(c\). Figure 4.14 shows the arithmetic compound assignment operators, sample expressions using the operators and explanations of what the operators do.

<table>
<thead>
<tr>
<th>Assignment operator</th>
<th>Sample expression</th>
<th>Explanation</th>
<th>Assigns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assume: int (c = 3, \ d = 5, \ e = 4, \ f = 6, \ g = 12;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>(c += 7)</td>
<td>(c = c + 7)</td>
<td>10 to (c)</td>
</tr>
<tr>
<td>-=</td>
<td>(d -= 4)</td>
<td>(d = d - 4)</td>
<td>1 to (d)</td>
</tr>
<tr>
<td>*=</td>
<td>(e *= 5)</td>
<td>(e = e * 5)</td>
<td>20 to (e)</td>
</tr>
</tbody>
</table>

Fig. 4.13 | Test program for class Analysis (Fig. 4.12). (Part 2 of 2.)

Fig. 4.14 | Arithmetic compound assignment operators. (Part 1 of 2.)
### 4.12 Increment and Decrement Operators

Java provides two unary operators for adding 1 to or subtracting 1 from the value of a numeric variable. These are the unary increment operator, `++`, and the unary decrement operator, `--`, which are summarized in Fig. 4.15. A program can increment by 1 the value of a variable called `c` using the increment operator, `++`, rather than the expression `c = c + 1` or `c += 1`. An increment or decrement operator that is prefixed to (placed before) a variable is referred to as the *prefix increment* or *prefix decrement operator*, respectively. An increment or decrement operator that is postfixed to (placed after) a variable is referred to as the *postfix increment* or *postfix decrement operator*, respectively.

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

**Good Programming Practice 4.7**

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

### Figure 4.15

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator name</th>
<th>Sample expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++</code></td>
<td>prefix increment</td>
<td><code>++a</code></td>
<td>Increment <code>a</code> by 1, then use the new value of <code>a</code> in the expression in which <code>a</code> resides.</td>
</tr>
<tr>
<td><code>++</code></td>
<td>postfix increment</td>
<td><code>a++</code></td>
<td>Use the current value of <code>a</code> in the expression in which <code>a</code> resides, then increment <code>a</code> by 1.</td>
</tr>
<tr>
<td><code>--</code></td>
<td>prefix decrement</td>
<td><code>--b</code></td>
<td>Decrement <code>b</code> by 1, then use the new value of <code>b</code> in the expression in which <code>b</code> resides.</td>
</tr>
<tr>
<td><code>--</code></td>
<td>postfix decrement</td>
<td><code>b--</code></td>
<td>Use the current value of <code>b</code> in the expression in which <code>b</code> resides, then decrement <code>b</code> by 1.</td>
</tr>
</tbody>
</table>
Figure 4.16 demonstrates the difference between the prefix increment and postfix increment versions of the ++ increment operator. The decrement operator (--) works similarly. Note that this example contains only one class, with method main performing all the class’s work. In this chapter and in Chapter 3, you have seen examples consisting of two classes—one class containing methods that perform useful tasks and one containing method main, which creates an object of the other class and calls its methods. In this example, we simply want to show the mechanics of the ++ operator, so we use only one class declaration containing method main. Occasionally, when it does not make sense to try to create a reusable class to demonstrate a simple concept, we will use a “mechanical” example contained entirely within the main method of a single class.

Line 11 initializes the variable c to 5, and line 12 outputs c’s initial value. Line 13 outputs the value of the expression c++. This expression postincrements the variable c, so c’s original value (5) is output, then c’s value is incremented (to 6). Thus, line 13 outputs c’s

```java
// Fig. 4.16: Increment.java
// Prefix increment and postfix increment operators.

public class Increment
{
    public static void main( String args[] )
    {
        int c;

        // demonstrate postfix increment operator
        c = 5; // assign 5 to c
        System.out.println( c ); // prints 5
        System.out.println( c++ ); // prints 5 then postincrements
        System.out.println( c ); // prints 6
        System.out.println(); // skip a line

        // demonstrate prefix increment operator
        c = 5; // assign 5 to c
        System.out.println( c ); // prints 5
        System.out.println( ++c ); // preincrements then prints 6
        System.out.println( c ); // prints 6
    }
}
```

Fig. 4.16 | Preincrementing and postincrementing.
initial value (5) again. Line 14 outputs c’s new value (6) to prove that the variable’s value was indeed incremented in line 13.

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

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

\[
\begin{align*}
\text{passes} & = \text{passes} + 1; \\
\text{failures} & = \text{failures} + 1; \\
\text{studentCounter} & = \text{studentCounter} + 1;
\end{align*}
\]

can be written more concisely with compound assignment operators as

\[
\begin{align*}
\text{passes} & += 1; \\
\text{failures} & += 1; \\
\text{studentCounter} & += 1;
\end{align*}
\]

with prefix increment operators as

\[
\begin{align*}
++\text{passes}; \\
++\text{failures}; \\
++\text{studentCounter};
\end{align*}
\]

or with postfix increment operators as

\[
\begin{align*}
\text{passes}++; \\
\text{failures}++; \\
\text{studentCounter}++; \\
\end{align*}
\]

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

**Common Programming Error 4.9**

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

Figure 4.17 shows the precedence and associativity of the operators we have introduced 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. The conditional operator (?:); the unary operators increment (++) and decrement (--), plus (+) and minus (-); the cast operators and the assignment operators =, +=, -=, *=, /= and %= associate from right to left. All the other operators in the operator precedence chart in Fig. 4.17 associate from left to right. The third column lists the type of each group of operators.
4.13 Primitive Types

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

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

**Portability Tip 4.1**

> Unlike C and C++, the primitive types in Java are portable across all computer platforms that support Java.

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

Recall from Section 3.5 that variables of primitive types declared outside of a method as fields of a class are automatically assigned default values unless explicitly initialized. Instance variables of types char, byte, short, int, long, float and double are all given the value 0 by default. Instance variables of type boolean are given the value false by default. Reference-type instance variables are initialized by default to the value null.

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

One of Java’s appealing features is its graphics support that enables programmers to visually enhance their applications. This section introduces one of Java’s graphical capabi-
tries—drawing lines. It also covers the basics of creating a window to display a drawing on the computer screen.

To draw in Java, you must understand Java’s coordinate system (Fig. 4.18), a scheme for identifying every point on the screen. By default, the upper-left corner of a GUI component has the coordinates (0, 0). A coordinate pair is composed of an x-coordinate (the horizontal coordinate) and a y-coordinate (the vertical coordinate). The x-coordinate is the horizontal location moving from left to right. The y-coordinate is the vertical location moving top to bottom. The x-axis describes every horizontal coordinate, and the y-axis every vertical coordinate.

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

Our first drawing application simply draws two lines. Class DrawPanel (Fig. 4.19) performs the actual drawing, while class DrawPanelTest (Fig. 4.20) creates a window to display the drawing. In class DrawPanel, the import statements in lines 3–4 allow us to use class Graphics (from package java.awt), which provides various methods for drawing text and shapes onto the screen, and class JPanel (from package javax.swing), which provides an area on which we can draw.

![Java coordinate system. Units are measured in pixels.](image)

```java
1 // Fig. 4.19: DrawPanel.java
2 // Using drawLine to connect the corners of a panel.
3 import java.awt.Graphics;
4 import javax.swing.JPanel;
5
6 public class DrawPanel extends JPanel
7 {
8     // draws an X from the corners of the panel
9     public void paintComponent( Graphics g )
10     {
11         // call paintComponent to ensure the panel displays correctly
12         super.paintComponent( g );
13     }
```

Fig. 4.19 Using drawLine to connect the corners of a panel. (Part 1 of 2.)
4.14 (Optional) GUI and Graphics Case Study: Creating Simple Drawings

```java
14     int width = getWidth(); // total width
15     int height = getHeight(); // total height
16     // draw a line from the upper-left to the lower-right
17     g.drawLine( 0, 0, width, height );
18     // draw a line from the lower-left to the upper-right
19     g.drawLine( height, width, 0 );
20 } // end method paintComponent
21 } // end class DrawPanel
```

Fig. 4.19 | Using drawLine to connect the corners of a panel. (Part 2 of 2.)

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

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

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

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

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

To display the `DrawPanel` on the screen, we must place it in a window. You create a window with an object of class `JFrame`. In `DrawPanelTest.java` (Fig. 4.20), line 3 imports class `JFrame` from package `javax.swing`. Line 10 in the main method of class `DrawPanelTest` creates an instance of class `DrawPanel`, which contains our drawing, and line
// Fig. 4.20: DrawPanelTest.java
// Application to display a DrawPanel.
import javax.swing.JFrame;

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

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

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

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

Fig. 4.20 | Creating JFrame to display DrawPanel.

13 creates a new JFrame that can hold and display our panel. Line 16 calls method setDefaultCloseOperation with the argument JFrame.EXIT_ON_CLOSE to indicate that the application should terminate when the user closes the window. Line 18 uses JFrame’s add method to attach the DrawPanel containing our drawing to the JFrame. Line 19 sets the size of the JFrame. Method setSize takes two parameters—the width of the JFrame, and the height. Finally, line 20 displays the JFrame. When the JFrame is displayed, the DrawPanel’s paintComponent method (lines 9–22 of Fig. 4.19) is called, and the two lines are drawn (see the sample outputs in Fig. 4.20). Try resizing the window to see that the lines always draw based on the window’s current width and height.

GUI and Graphics Case Study Exercises

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

a) Create the design in the left screen capture of Fig. 4.21. This design draws lines from the top-left corner, fanning out the lines until they cover the upper-left half of the panel. One approach is to divide the width and height into an equal number of steps (we found
15 steps worked well). The first endpoint of a line will always be in the top-left corner (0, 0). The second endpoint can be found by starting at the bottom-left corner and moving up one vertical step and right one horizontal step. Draw a line between the two endpoints. Continue moving up and to the right one step to find each successive endpoint. The figure should scale accordingly as you resize the window.

b) Modify your answer in part (a) to have lines fan out from all four corners, as shown in the right screen capture of Fig. 4.21. Lines from opposite corners should intersect along the middle.

4.2 Figure 4.22 displays two additional designs created using while loops and drawLine.

a) Create the design in the left screen capture of Fig. 4.22. Begin by dividing each edge into an equal number of increments (we chose 15 again). The first line starts in the top-left corner and ends one step right on the bottom edge. For each successive line, move down one increment on the left edge and right one increment on the bottom edge. Continue drawing lines until you reach the bottom-right corner. The figure should scale as you resize the window so that the endpoints always touch the edges.

b) Modify your answer in part (a) to mirror the design in all four corners, as shown in the right screen capture of Fig. 4.22.

Fig. 4.21  |  Lines fanning from a corner.

Fig. 4.22  |  Line art with loops and drawLine.
4.15 (Optional) Software Engineering Case Study: Identifying Class Attributes

In Section 3.10, we began the first stage of an object-oriented design (OOD) for our ATM system—analyzing the requirements document and identifying the classes needed to implement the system. We listed the nouns and noun phrases in the requirements document and identified a separate class for each one that plays a significant role in the ATM system. We then modeled the classes and their relationships in a UML class diagram (Fig. 3.24). Classes have attributes (data) and operations (behaviors). Class attributes are implemented in Java programs as fields, and class operations are implemented as methods. In this section, we determine many of the attributes needed in the ATM system. In Chapter 5, we examine how these attributes represent an object’s state. In Chapter 6, we determine class operations.

Identifying Attributes

Consider the attributes of some real-world objects: A person’s attributes include height, weight and whether the person is left-handed, right-handed or ambidextrous. A radio’s attributes include its station setting, its volume setting and its AM or FM setting. A car’s attributes include its speedometer and odometer readings, the amount of gas in its tank and what gear it is in. A personal computer’s attributes include its manufacturer (e.g., Dell, Sun, Apple or IBM), type of screen (e.g., LCD or CRT), main memory size and hard disk size.

We can identify many attributes of the classes in our system by looking for descriptive words and phrases in the requirements document. For each one we find that plays a significant role in the ATM system, we create an attribute and assign it to one or more of the classes identified in Section 3.10. We also create attributes to represent any additional data that a class may need, as such needs become clear throughout the design process.

Figure 4.23 lists the words or phrases from the requirements document that describe each class. We formed this list by reading the requirements document and identifying any words or phrases that refer to characteristics of the classes in the system. For example, the requirements document describes the steps taken to obtain a “withdrawal amount,” so we list “amount” next to class Withdrawal.

<table>
<thead>
<tr>
<th>Class</th>
<th>Descriptive words and phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>user is authenticated</td>
</tr>
<tr>
<td>BalanceInquiry</td>
<td>account number</td>
</tr>
<tr>
<td>Withdrawal</td>
<td>account number</td>
</tr>
<tr>
<td></td>
<td>amount</td>
</tr>
<tr>
<td>Deposit</td>
<td>account number</td>
</tr>
<tr>
<td></td>
<td>amount</td>
</tr>
<tr>
<td>BankDatabase</td>
<td>[no descriptive words or phrases]</td>
</tr>
</tbody>
</table>

Fig. 4.23 | Descriptive words and phrases from the ATM requirements. (Part 1 of 2.)
### 4.15 Identifying Class Attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Descriptive words and phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>account number, PIN, balance</td>
</tr>
<tr>
<td>Screen</td>
<td>[no descriptive words or phrases]</td>
</tr>
<tr>
<td>Keypad</td>
<td>[no descriptive words or phrases]</td>
</tr>
<tr>
<td>CashDispenser</td>
<td>begins each day loaded with 500 $20 bills</td>
</tr>
<tr>
<td>DepositSlot</td>
<td>[no descriptive words or phrases]</td>
</tr>
</tbody>
</table>

**Fig. 4.23** | Descriptive words and phrases from the ATM requirements. (Part 2 of 2.)

Figure 4.23 leads us to create one attribute of class ATM. Class ATM maintains information about the state of the ATM. The phrase “user is authenticated” describes a state of the ATM (we introduce states in Section 5.11), so we include `userAuthenticated` as a **Boolean attribute** (i.e., an attribute that has a value of either `true` or `false`) in class ATM. Note that the Boolean attribute type in the UML is equivalent to the boolean type in Java. This attribute indicates whether the ATM has successfully authenticated the current user—`userAuthenticated` must be `true` for the system to allow the user to perform transactions and access account information. This attribute helps ensure the security of the data in the system.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share one attribute. Each transaction involves an “account number” that corresponds to the account of the user making the transaction. We assign an integer attribute `accountNumber` to each transaction class to identify the account to which an object of the class applies.

Descriptive words and phrases in the requirements document also suggest some differences in the attributes required by each transaction class. The requirements document indicates that to withdraw cash or deposit funds, users must input a specific “amount” of money to be withdrawn or deposited, respectively. Thus, we assign to classes `Withdrawal` and `Deposit` an attribute `amount` to store the value supplied by the user. The amounts of money related to a withdrawal and a deposit are defining characteristics of these transactions that the system requires for these transactions to take place. Class `BalanceInquiry`, however, needs no additional data to perform its task—it requires only an account number to indicate the account whose balance should be retrieved.

Class `Account` has several attributes. The requirements document states that each bank account has an “account number” and “PIN,” which the system uses for identifying accounts and authenticating users. We assign to class `Account` two integer attributes: `accountNumber` and `pin`. The requirements document also specifies that an account maintains a “balance” of the amount of money in the account and that money the user deposits does not become available for a withdrawal until the bank verifies the amount of cash in the deposit envelope, and any checks in the envelope clear. An account must still record the amount of money that a user deposits, however. Therefore, we decide that an account should represent a balance using two attributes: `availableBalance` and `totalBalance`. 
Attribute availableBalance tracks the amount of money that a user can withdraw from the account. Attribute totalBalance refers to the total amount of money that the user has “on deposit” (i.e., the amount of money available, plus the amount waiting to be verified or cleared). For example, suppose an ATM user deposits $50.00 into an empty account. The totalBalance attribute would increase to $50.00 to record the deposit, but the availableBalance would remain at $0. [Note: We assume that the bank updates the availableBalance attribute of an Account some length of time after the ATM transaction occurs, in response to confirming that $50 worth of cash or checks was found in the deposit envelope. We assume that this update occurs through a transaction that a bank employee performs using some piece of bank software other than the ATM. Thus, we do not discuss this transaction in our case study.]

Class CashDispenser has one attribute. The requirements document states that the cash dispenser “begins each day loaded with 500 $20 bills.” The cash dispenser must keep track of the number of bills it contains to determine whether enough cash is on hand to satisfy withdrawal requests. We assign to class CashDispenser an integer attribute count, which is initially set to 500.

For real problems in industry, there is no guarantee that requirements documents will be rich enough and precise enough for the object-oriented systems designer to determine all the attributes or even all the classes. The need for additional classes, attributes and behaviors may become clear as the design process proceeds. As we progress through this case study, we too will continue to add, modify and delete information about the classes in our system.

**Modeling Attributes**

The class diagram in Fig. 4.24 lists some of the attributes for the classes in our system—the descriptive words and phrases in Fig. 4.23 lead us to identify these attributes. For simplicity, Fig. 4.24 does not show the associations among classes—we showed these in Fig. 3.24. This is a common practice of systems designers when designs are being developed. Recall from Section 3.10 that in the UML, a class’s attributes are placed in the middle compartment of the class’s rectangle. We list each attribute’s name and type separated by a colon (:), followed in some cases by an equal sign (=) and an initial value.

Consider the userAuthenticated attribute of class ATM:

```
userAuthenticated : Boolean = false
```

This attribute declaration contains three pieces of information about the attribute. The attribute name is userAuthenticated. The attribute type is Boolean. In Java, an attribute can be represented by a primitive type, such as boolean, int or double, or a reference type like a class—as discussed in Chapter 3. We have chosen to model only primitive-type attributes in Fig. 4.24—we discuss the reasoning behind this decision shortly. [Note: The attribute types in Fig. 4.24 are in UML notation. We will associate the types boolean, Integer and double in the UML diagram with the primitive types boolean, int and double in Java, respectively.]

We can also indicate an initial value for an attribute. The userAuthenticated attribute in class ATM has an initial value of false. This indicates that the system initially does not consider the user to be authenticated. If an attribute has no initial value specified, only its name and type (separated by a colon) are shown. For example, the accountNumber
attribute of class **BalanceInquiry** is an integer. 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 at execution time based on the account number entered by the current ATM user.

Figure 4.24 does not include any attributes for classes **Screen**, **Keypad** and **DepositSlot**. These are important components of our system, for which our design process simply has not yet revealed any attributes. We may still discover some, however, in the remaining phases of design or when we implement these classes in Java. This is perfectly normal.

**Software Engineering Observation 4.6**

At early stages in the design process, classes often lack attributes (and operations). Such classes should not be eliminated, however, because attributes (and operations) may become evident in the later phases of design and implementation.

Note that Fig. 4.24 also does not include attributes for class **BankDatabase**. Recall from Chapter 3 that in Java, attributes can be represented by either primitive types or reference types. We have chosen to include only primitive-type attributes in the class diagram in Fig. 4.24 (and in similar class diagrams throughout the case study). A reference-type attribute is modeled more clearly as an association (in particular, a composition) between the class holding the reference and the class of the object to which the reference points.
For example, the class diagram in Fig. 3.24 indicates that class BankDatabase participates in a composition relationship with zero or more Account objects. From this composition, we can determine that when we implement the ATM system in Java, we will be required to create an attribute of class BankDatabase to hold references to zero or more Account objects. Similarly, we can determine reference-type attributes of class ATM that correspond to its composition relationships with classes Screen, Keypad, CashDispenser and DepositSlot. These composition-based attributes would be redundant if modeled in Fig. 4.24, because the compositions modeled in Fig. 3.24 already convey the fact that the database contains information about zero or more accounts and that an ATM is composed of a screen, keypad, cash dispenser and deposit slot. Software developers typically model these whole/part relationships as compositions rather than as attributes required to implement the relationships.

The class diagram in Fig. 4.24 provides a solid basis for the structure of our model, but the diagram is not complete. In Section 5.11, we identify the states and activities of the objects in the model, and in Section 6.14 we identify the operations that the objects perform. As we present more of the UML and object-oriented design, we will continue to strengthen the structure of our model.

**Software Engineering Case Study Self-Review Exercises**

4.1 We typically identify the attributes of the classes in our system by analyzing the _______ in the requirements document.

   a) nouns and noun phrases
   b) descriptive words and phrases
   c) verbs and verb phrases
   d) All of the above.

4.2 Which of the following is not an attribute of an airplane?

   a) length
   b) wingspan
   c) fly
   d) number of seats

4.3 Describe the meaning of the following attribute declaration of class CashDispenser in the class diagram in Fig. 4.24:

   count : Integer = 500

**Answers to Software Engineering Case Study Self-Review Exercises**

4.1 b.

4.2 c. Fly is an operation or behavior of an airplane, not an attribute.

4.3 This indicates that count is an Integer with an initial value of 500. This attribute keeps track of the number of bills available in the CashDispenser at any given time.

**4.16 Wrap-Up**

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

Only three types of control structures—sequence, selection and repetition—are needed to develop any problem-solving algorithm. Specifically, this chapter demonstrated the if single-selection statement, the if...else double-selection statement and the while repetition statement. These are some of the building blocks used to construct solutions to many problems. We used control-statement stacking to total and compute the average of a set of student grades with counter- and sentinel-controlled repetition, and we used control-statement nesting to analyze and make decisions based on a set of exam results. We introduced Java’s compound assignment operators, and its increment and decrement operators. Finally, we discussed the primitive types available to Java programmers. In Chapter 5, Control Statements: Part 2, we continue our discussion of control statements, introducing the for, do...while and switch statements.

Summary

Section 4.1 Introduction
• Before writing a program to solve a problem, you must have a thorough understanding of the problem and a carefully planned approach to solving it. You must also understand the building blocks that are available and to employ proven program-construction techniques.

Section 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 the actions to execute and the order in which they execute is called an algorithm.
• Specifying the order in which statements execute in a program is called program control.

Section 4.3 Pseudocode
• Pseudocode is an informal language that helps programmers develop algorithms without having to worry about the strict details of Java language syntax.
• Pseudocode is similar to everyday English—it is convenient and user friendly, but it is not an actual computer programming language.
• Pseudocode helps the programmer “think out” a program before attempting to write it in a programming language, such as Java.
• Carefully prepared pseudocode can easily be converted to a corresponding Java program.

Section 4.4 Control Structures
• Normally, statements in a program are executed one after the other in the order in which they are written. This process is called sequential execution.
• Various Java statements enable the programmer to specify that the next statement to execute is not necessarily the next one in sequence. This is called transfer of control.
• Bohm and Jacopini demonstrated that all programs could be written in terms of only three control structures—the sequence structure, the selection structure and the repetition structure.
• The term “control structures” comes from the field of computer science. The Java Language Specification refers to “control structures” as “control statements.”
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—that is, in sequence.

--

The arrows in an activity diagram represent transitions, which indicate the order in which the actions represented by the action states occur.

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

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

Rectangles with their upper-right corners folded over are UML notes—explanatory remarks that describe the purpose of symbols in the diagram.

Java has three types of selection statements. The if statement either performs an action if a condition is true or skips the action if the condition is false. The if...else statement performs an action if a condition is true and a different action if the condition is false. The switch statement performs one of many different actions, depending on the value of an expression.

The if statement is a single-selection statement because it selects or ignores a single action or a single group of actions.

The if...else statement is called a double-selection statement because it selects between two different actions or groups of actions.

The switch statement is called a multiple-selection statement because it selects among many different actions or groups of actions.

Java provides the the while, do...while and for repetition (looping) statements that enable programs to perform statements repeatedly as long as a loop-continuation condition remains true.

The while and for statements perform the action(s) in their bodies zero or more times—if the loop-continuation condition is initially false, the action(s) will not execute. The do...while statement performs the action(s) in its body one or more times.

The words if, else, switch, while, do and for are Java keywords. Keywords cannot be used as identifiers, such as variable names.

Every program is formed by combining as many sequence, selection and repetition statements as is appropriate for the algorithm the program implements.

Single-entry/single-exit control statements make it easy to build programs—we “attach” the control statements to one another by connecting the exit point of one to the entry point of the next. This is known as control-statement stacking.

There is only one other way in which control statements may be connected—control-statement nesting—in which a control statement appears inside another control statement.

**Section 4.5 if Single-Selection Statement**

Programs use selection statements to choose among alternative courses of action.
• The activity diagram for the single-selection if statement contains the diamond, or decision symbol, which indicates that a decision is to be made. The workflow will continue along a path determined by the symbol’s associated guard conditions, which can be true or false. Each transition arrow emerging from a decision symbol has a guard condition. If a guard condition is true, the workflow enters the action state to which the transition arrow points.
• The if statement is a single-entry/single-exit control statement.

Section 4.6 if...else Double-Selection Statement
• The if single-selection statement performs an indicated action only when the condition is true.
• The if...else double-selection statement performs one action when the condition is true and a different action when the condition is false.
• The conditional operator (?:) can be used in place of an if...else statement. This is Java’s only ternary operator—it takes three operands. Together, the operands and the ?: symbol form a conditional expression.
• A program can test multiple cases by placing if...else statements inside other if...else statements to create nested if...else statements.
• The Java compiler always associates an else with the immediately preceding if unless told to do otherwise by the placement of braces ({ and }). This behavior can lead to what is referred to as the dangling-else problem.
• The if statement normally expects only one statement in its body. To include several statements in the body of an if (or the body of an else for an if...else statement), enclose the statements in braces ({ and }).
• A set of statements contained within a pair of braces is called a block. A block can be placed anywhere in a program that a single statement can be placed.
• Syntax errors are caught by the compiler.
• A logic error has its effect at execution time. A fatal logic error causes a program to fail and terminate prematurely. A nonfatal logic error allows a program to continue executing, but causes the program to produce incorrect results.
• Just as a block can be placed anywhere a single statement can be placed, you can also use an empty statement, represented by placing a semicolon (;) where a statement would normally be.

Section 4.7 while Repetition Statement
• The while repetition statement allows the programmer to specify that a program should repeat an action while some condition remains true.
• The UML’s merge symbol joins two flows of activity into one.
• The decision and merge symbols can be distinguished by the number of “incoming” and “outgoing” transition arrows. A decision symbol has one transition arrow pointing to the diamond and two or more transition arrows pointing out from the diamond to indicate possible transitions from that point. Each transition arrow pointing out of a decision symbol has a guard condition. A merge symbol has two or more transition arrows pointing to the diamond and only one transition arrow pointing from the diamond, to indicate multiple activity flows merging to continue the activity. None of the transition arrows associated with a merge symbol has a guard condition.

Section 4.8 Formulating Algorithms: Counter-Controlled Repetition
• Counter-controlled repetition uses a variable called a counter (or control variable) to control the number of times a set of statements execute.
• Counter-controlled repetition is often called definite repetition, because the number of repetitions is known before the loop begins executing.
• A total is a variable used to accumulate the sum of several values. Variables used to store totals are normally initialized to zero before being used in a program.
• A local variable’s declaration must appear before the variable is used in that method. A local variable cannot be accessed outside the method in which it is declared.
• Dividing two integers results in integer division—the calculation’s fractional part is truncated.

Section 4.9 Formulating Algorithms: Sentinel-Controlled Repetition
• In sentinel-controlled repetition, a special value called a sentinel value (also called a signal value, a dummy value or a flag value) is used to indicate “end of data entry.”
• A sentinel value must be chosen that cannot be confused with an acceptable input value.
• Top-down, stepwise refinement is essential to the development of well-structured programs.
• Division by zero is normally a logic error that, if undetected, would cause the program to fail or produce invalid output.
• To perform a floating-point calculation with integer values, cast one of the integers to type double. Using a cast operator in this manner is called explicit conversion.
• Java knows how to evaluate only arithmetic expressions in which the operands’ types are identical. To ensure that the operands are of the same type, Java performs an operation called promotion (or implicit conversion) on selected operands. In an expression containing values of the types int and double, the int values are promoted to double values for use in the expression.
• Cast operators are available for any type. The cast operator is formed by placing parentheses around the name of a type. The operator is a unary operator.

Section 4.11 Compound Assignment Operators
• Java provides several compound assignment operators for abbreviating assignment expressions. Any statement of the form

\[
\text{variable } = \text{variable } \text{operator expression;}
\]

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

\[
\text{variable } \text{operator=} \text{ expression;}
\]

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

Section 4.12 Increment and Decrement Operators
• Java provides two unary operators for adding 1 to or subtracting 1 from the value of a numeric variable. These are the unary increment operator, ++, and the unary decrement operator, --.
• An increment or decrement operator that is prefixed to a variable is the prefix increment or prefix decrement operator, respectively. An increment or decrement operator that is postfixed to a variable is the postfix increment or postfix decrement operator, respectively.
• Using the prefix increment or decrement operator to add or subtract 1 is known as preincrementing or predecrementing, respectively.
• Preincrementing or predecrementing a variable causes the variable to be incremented or decremented by 1, and then the new value of the variable is used in the expression in which it appears.
• Using the postfix increment or decrement operator to add or subtract 1 is known as postincrementing or postdecrementing, respectively.
• Postincrementing or postdecrementing the variable causes the current value of the variable to be used in the expression in which it appears, and then the variable’s value is incremented or decremented by 1.
• When incrementing or decrementing a variable in a statement by itself, the prefix and postfix increment forms have the same effect, and the prefix and postfix decrement forms have the same effect.

Section 4.13 Primitive Types
• Java requires all variables to have a type. Thus, Java is referred to as a strongly typed language.
• Because the designers of Java want it to be maximally portable, they use internationally recognized standards for both character formats (Unicode) and floating-point numbers (IEEE 754).

Section 4.14 (Optional) GUI and Graphics Case Study: Creating Simple Drawings
• Java’s coordinate system provides a scheme for identifying every point on the screen. By default, the upper-left corner of a GUI component has the coordinates (0, 0).
• A coordinate pair is composed of an x-coordinate (the horizontal coordinate) and a y-coordinate (the vertical coordinate). The x-coordinate is the horizontal location moving from left to right. The y-coordinate is the vertical location moving top to bottom.
• The x-axis describes every horizontal coordinate, and the y-axis every vertical coordinate.
• Coordinate units are measured in pixels. A pixel is a display monitor’s smallest unit of resolution.
• Class Graphics (from package java.awt) provides various methods for drawing text and shapes onto the screen.
• Class JPanel (from package javax.swing) provides an area on which a program can draw.
• The keyword extends indicates that a class inherits from another class. The new class begins with the existing members (data and methods) from the existing class.
• The class from which the new class inherits is called the superclass and the new class is called the subclass.
• Every JPanel has a paintComponent method, which the system automatically calls every time it needs to display the JPanel—when a JPanel is first displayed on the screen, when it is covered then uncovered by a window on the screen and when the window in which it appears is resized.
• Method paintComponent requires one argument, a Graphics object, that is provided for you by the system when it calls paintComponent.
• The first statement in every paintComponent method you create should always be

  super.paintComponent( g );

  This ensures that the panel is properly rendered on the screen before you begin drawing on it.
• JPanel methods getWidth and getHeight return the width and height of a JPanel, respectively.
• Graphics method drawLine draws a line between two points represented by its four arguments. The first two arguments are the x- and y-coordinates for one endpoint of the line, and the last two arguments are the coordinates for the other endpoint of the line.
• To display a JPanel on the screen, you must place it in a window. You create a window with an object of class JFrame from package javax.swing.
• JFrame method setDefaultCloseOperation with the argument JFrame.EXIT_ON_CLOSE indicates that the application should terminate when the user closes the window.
• JFrame method add attaches a GUI component to a JFrame.
• JFrame method setSize sets the width and height of the JFrame.

**Terminology**

++ operator
++ operator
-- operator
?: operator
action
action/decision model of programming
action expression (in the UML)
action state (in the UML)
action-state symbol (in the UML)
activity (in the UML)
activity diagram (in the UML)
add method of class JFrame (GUI)
addition compound assignment operator (++)
algorithm
arithmetic compound assignment operators: 
\[+\lt, -\lt, \ast\lt, /\lt \text{ and } \%\lt\]
block
boolean expression
boolean primitive type
body of a loop
cast operator, (type)
compound assignment operator
conditional expression
conditional operator (?:)
control statement
control-statement nesting
control-statement stacking
control variable
coordinate system (GUI)
counter
counter-controlled repetition
dangling-else problem
decision
decision symbol (in the UML)
decrement operator (--)
definite repetition
diamond (in the UML)
dotted line (in the UML)
(double) cast operator
double-selection statement
drawLine method of class Graphics (GUI)
dummy value
explicit conversion
extends
false
fatal error

fatal logic error
final state (in the UML)
first refinement
flag value
getHeight method of class JPanel (GUI)
getWidth method of class JPanel (GUI)
goto statement
Graphics class (GUI)
guard condition (in the UML)
horizontal coordinate (GUI)
if single-selection statement
if...else double-selection statement
implicit conversion
increment operator (++)
indeterminate repetition
infinite loop
inherit from an existing class
initial state (in the UML)
initialization
integer division
iteration
JComponent class (GUI)
JFrame class (GUI)
JPanel class (GUI)
logic error
loop
loop-continuation condition
loop counter
looping statement
merge symbol (in the UML)
multiple-selection statement
multiplicative operator
nested control statements
nested if...else statements
nonfatal logic error
note (in the UML)
order in which actions should execute
paintComponent method of class JComponent (GUI)
pixel (GUI)
postdecrement a variable
postfix decrement operator
postfix increment operator
postincrement a variable
predecrement a variable
prefix decrement operator
Self-Review Exercises

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

4.2 State whether each of the following is true or false. If false, explain why.
   a) An algorithm is a procedure for solving a problem in terms of the actions to execute and the order in which they execute.
   b) A set of statements contained within a pair of parentheses is called a block.
   c) A selection statement specifies that an action is to be repeated while some condition remains true.
Chapter 4  Control Statements: Part 1

d) A nested control statement appears in the body of another control statement.
c) Java provides the arithmetic compound assignment operators +=, -=, *=, /= and %= for abbreviating assignment expressions.
f) The primitive types (boolean, char, byte, short, int, long, float and double) are portable across only Windows platforms.
g) Specifying the order in which statements (actions) execute in a program is called program control.
h) The unary cast operator (double) creates a temporary integer copy of its operand.
i) Instance variables of type boolean are given the value true by default.
j) Pseudocode helps a programmer think out a program before attempting to write it in a programming language.

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

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

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

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

4.7 Determine the value of the variables in the following statement after the calculation is performed. Assume that when the statement begins executing, all variables are type int and have the value 5.

    product *= x++;

4.8 Identify and correct the errors in each of the following sets of code:
a) while ( c <= 5 )
    {
      product *= c;
      ++c;
    }
b) if ( gender == 1 )
    System.out.println( "Woman" );
    else;
    System.out.println( "Man" );

4.9 What is wrong with the following while statement?

    while ( z >= 0 )
    sum += z;
Answers to Self-Review Exercises

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

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

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

4.4  
```java
a) z = x++ + y;
b) if ( count > 10 )
    System.out.println( "Count is greater than 10" );
c) total -= --x;
d) q %= divisor;
    q = q % divisor;
```

4.5  
```java
a) int sum, x;
b) x = 1;
c) sum = 0;
d) sum += x; or sum = sum + x;
c) System.out.printf( "The sum is: %d\n", sum );
```

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

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

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

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

The sum is: 55
4.7 \hspace{1em} \text{product} = 25, \ x = 6

4.8 \hspace{1em} \begin{align*}
\text{a)} & \quad \text{Error: The closing right brace of the while statement’s body is missing.} \\
& \hspace{1em} \text{Correction: Add a closing right brace after the statement ++c;.)} \\
\text{b)} & \quad \text{Error: The semicolon after else results in a logic error. The second output statement} \\
& \hspace{1em} \text{will always be executed.} \\
& \hspace{1em} \text{Correction: Remove the semicolon after else.}
\end{align*}

4.9 \hspace{1em} \text{The value of the variable } z \text{ is never changed in the while statement. Therefore, if the loop-} \\
\text{continuation condition } (z > 0) \text{ is true, an infinite loop is created. To prevent an infinite loop from} \\
\text{occurring, } z \text{ must be decremented so that it eventually becomes less than 0.}

\textbf{Exercises}

4.10 \hspace{1em} \text{Compare and contrast the if single-selection statement and the while repetition statement.} \\
\text{How are these two statements similar? How are they different?}

4.11 \hspace{1em} \text{Explain what happens when a Java program attempts to divide one integer by another.} \\
\text{What happens to the fractional part of the calculation? How can a programmer avoid that outcome?}

4.12 \hspace{1em} \text{Describe the two ways in which control statements can be combined.}

4.13 \hspace{1em} \text{What type of repetition would be appropriate for calculating the sum of the first 100 posi-} \\
\text{tive integers? What type of repetition would be appropriate for calculating the sum of an arbitrary} \\
\text{number of positive integers? Briefly describe how each of these tasks could be performed.}

4.14 \hspace{1em} \text{What is the difference between preincrementing and postincrementing a variable?}

4.15 \hspace{1em} \text{Identify and correct the errors in each of the following pieces of code. [Note: There may be} \\
\text{more than one error in each piece of code.]} \\
\hspace{1em} \begin{align*}
\text{a)} & \quad \text{if ( age >= 65 )} \\
& \hspace{1em} \text{System.out.println( "Age greater than or equal to 65" );} \\
& \hspace{1em} \text{else} \\
& \hspace{1em} \text{System.out.println( "Age is less than 65" );} \\
\text{b)} & \quad \text{int x = 1, total;} \\
& \hspace{1em} \text{while ( x <= 10 )} \\
& \hspace{1em} \{ \\
& \hspace{2em} \text{total += x;} \\
& \hspace{2em} ++x; \\
& \hspace{1em} \} \\
\text{c)} & \quad \text{while ( x <= 100 )} \\
& \hspace{1em} \text{total += x;} \\
& \hspace{1em} ++x; \\
\text{d)} & \quad \text{while ( y > 0 )} \\
& \hspace{1em} \{ \\
& \hspace{2em} \text{System.out.println( y );} \\
& \hspace{2em} ++y; \\
\end{align*}

4.16 \hspace{1em} \text{What does the following program print?}

```java
1  public class Mystery
2  {
3      public static void main( String args[] )
4      {
5          int y;
6          int x = 1;
```
```java
    int total = 0;
    while ( x <= 10 )
    {
        y = x * x;
        System.out.println( y );
        total += y;
        ++x;
    } // end while
    System.out.printf("Total is %d\n", total);
} // end main
} // end class Mystery
```

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

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

4.17 Drivers are concerned with the mileage their automobiles get. One driver has kept track of several tankfuls of gasoline by recording the miles driven and gallons used for each tankful. Develop a Java application that will input the miles driven and gallons used (both as integers) for each 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 class Scanner and sentinel-controlled repetition to obtain the data from the user.

4.18 Develop a Java application that will determine whether any of several department-store customers has exceeded the credit limit on a charge account. For each customer, the following facts are available:
   a) account number
   b) balance at the beginning of the month
   c) total of all items charged by the customer this month
   d) total of all credits applied to the customer’s account this month
   e) allowed credit limit.

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

4.19 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 the items sold by each salesperson. The values of these items are as follows:

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>239.99</td>
</tr>
<tr>
<td>2</td>
<td>129.75</td>
</tr>
<tr>
<td>3</td>
<td>99.95</td>
</tr>
<tr>
<td>4</td>
<td>350.89</td>
</tr>
</tbody>
</table>

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.

4.20 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 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 class Scanner to input the data.

4.21 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 integers and determines and prints the largest integer. Your program should use at least the following three variables:
   a) counter: A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed).
   b) number: The integer most recently input by the user.
   c) largest: The largest number found so far.

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

<table>
<thead>
<tr>
<th>N</th>
<th>10*N</th>
<th>100*N</th>
<th>1000*N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>200</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>300</td>
<td>3000</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>400</td>
<td>4000</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>500</td>
<td>5000</td>
</tr>
</tbody>
</table>

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

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

4.25 What does the following program print?

```java
public class Mystery2 {
    public static void main( String args[] ) {
        int count = 1;
        while ( count <= 10 ) {
            System.out.println( count % 2 == 1 ? "####" : "+++++++" );
            ++count;
        } // end while
    } // end main
} // end class Mystery2
```
4.26 What does the following program print?

```java
public class Mystery3 {
    public static void main( String args[] ) {
        int row = 10;
        int column;
        while ( row >= 1 ) {
            column = 1;
            while ( column <= 10 ) {
                System.out.print( row % 2 == 1 ? "<" : ">" );
                ++column;
            } // end while
            --row;
            System.out.println();
        } // end while
    } // end main
} // end class Mystery3
```

4.27 *(Dangling-else Problem)* Determine the output for each of the given sets of code when \( x \) is 9 and \( y \) is 11 and when \( x \) is 11 and \( y \) is 9. Note that the compiler ignores the indentation in a Java program. Also, the Java compiler always associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{}`). On first glance, 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 the 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( "$$$$" );
    }

4.28 *(Another Dangling-else Problem)* Modify the given code to produce the output shown in each part of the problem. Use proper indentation techniques. Make no changes other than inserting braces and changing the indentation of the code. The compiler ignores indentation in a Java pro-
gram. 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.]

```java
if ( y == 8 )
if ( x == 5 )
System.out.println( "00000" );
else
System.out.println( "####" );
System.out.println( "$$$$" );
System.out.println( "&&&" );
```

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

```
00000
$$$$$
&&&
```

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

```
00000
```

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

```
00000
#####
```

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

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

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

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

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

4.32 Write an application that uses only the output statements

```java
System.out.print( "#" );
System.out.print( "#" );
System.out.println();
```

to display the checkerboard pattern that follows. Note that a `System.out.println` method call with no arguments causes the program to output a single newline character. [Hint: Repetition statements are required.]
4.33 Write an application that keeps displaying in the command window the multiples of the integer 2—namely, 2, 4, 8, 16, 32, 64, and so on. Your loop should not terminate (i.e., create an infinite loop). What happens when you run this program?

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

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

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

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

4.37 A company wants to transmit data over the telephone but is concerned that its phones may be tapped. It has asked you to write a program that will encrypt the data so that it may be transmitted more securely. All the data is transmitted as four-digit integers. Your application should read a four-digit integer entered by the user and encrypt it as follows: Replace each digit with the result of adding 7 to the digit and getting the remainder after dividing the new value by 10. Then swap the first digit with the third, and swap the second digit with the fourth. Then print the encrypted integer. Write a separate application that inputs an encrypted four-digit integer and decrypts it to form the original number.

4.38 The factorial of a nonnegative integer \( n \) is written as \( n! \) (pronounced "\( n \) factorial") and is defined as follows:

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

and

\[
 n! = 1 \quad \text{for} \quad n = 0
\]

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

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

b) Write an application that estimates the value of the mathematical constant \( e \) by using the formula

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

c) Write an application that computes the value of \( e^x \) by using the formula

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