# 4.11 Formulating Algorithms: Nested Control Statements

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

Consider the following problem statement:

*A college offers a course that prepares students for the state licensing exam for real-estate brokers. Last year, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.*

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

1. *Input each test result (i.e., a 1 or a 2). Display the message "Enter result"*

*on the screen each time the program requests another test result.*

2. *Count the number of test results of each type.*
3. *Display a summary of the test results, indicating the number of students who passed and the number who failed.*
4. *If more than eight students passed the exam, print "Bonus to instructor!"*

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

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

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

```
Analyze exam results and decide whether a bonus shoul
```

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 failu
Print a summary of the exam results and decide whethe
```

Here, too, even though we have a *complete* representation of the 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 input 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 failu
```

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 whethe
```

can be refined as follows:

```
Print the number of passes
Print the number of failures
If more than eight students passed
    Print “Bonus to instructor!”
```

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

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

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

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

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

   Add one to student counter

```

```
16 Print the number of passes
17 Print the number of failures
18
19 If more than eight students passed
20    Print “Bonus to instructor!”
```

# Fig. 4.11

Pseudocode for examination-results problem.

The Java class that implements the pseudocode algorithm and two sample executions are shown in Fig. 4.12. Lines 11–13 and 19 of `main` declare the variables that are used to process the examination results.

## Error-Prevention Tip 4.5

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

The `while` statement (lines 16–31) loops 10 times. During each iteration, the loop inputs and processes one exam result. Notice that the `if`…`else` statement (lines 22–27) 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 30 increments `studentCounter` before the loop condition is tested again at line 16. After 10 values have been input, the loop terminates and line 34 displays the number of `passes` and `failures`. The `if` statement at lines 37–39 determines whether more than eight students passed the exam and, if so, outputs the message `"Bonus to instructor!"`.

Figure 4.12 shows the input and output from two sample executions of the program. During the first, the condition at line 37 of method `main` is `true`—more than eight students passed the exam, so the program outputs a message to bonus the instructor.

```
          1 // Fig. 4.12: Analysis.java
 2 // Analysis of examination results using nested co
 3 import java.util.Scanner; // class uses class Scan
                        4
           5 public class Analysis {
  6    public static void main(String[] args) {
 7       // create Scanner to obtain input from comma
 8       Scanner input = new Scanner(System.in);
                        9
10       // initializing variables in declarations
            11       int passes = 0;
           12       int failures = 0;
         13       int studentCounter = 1;
                       14
15       // process 10 students using counter-control
     16       while (studentCounter <= 10) {
17          // prompt user for input and obtain value
```

```
18          System.out.print("Enter result (1 = pass,
    19          int result = input.nextInt();
                    20
21          // if …else is nested in the while statem
         22          if (result == 1) {
       23             passes = passes + 1;
                24          }
              25          else {
     26             failures = failures + 1;
                27          }
                    28
29          // increment studentCounter so loop event
30          studentCounter = studentCounter + 1;
                   31       }
                    32
33       // termination phase; prepare and display re
34       System.out.printf("Passed: %d%nFailed: %d%n"
                    35
36       // determine whether more than 8 students pa
           37       if (passes > 8) {
38          System.out.println("Bonus to instructor!"
                   39       }
                    40    }
                      41 }
```

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

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

# Fig. 4.12

Analysis of examination results using nested control statements.