4.10 Formulating Algorithms: Sentinel-Controlled Iteration

Let’s generalize Section 4.9’s class-average problem. Consider the following problem:

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

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

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

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

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

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

```
Determine the class average for the quiz
```

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

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

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

Software Engineering Observation 4.3

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

Software Engineering Observation 4.4

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

Proceeding to the Second Refinement

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

```
Initialize variables
```

can be refined as follows:

```
Initialize total to zero
Initialize counter to zero
```

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

The pseudocode statement

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

requires *iteration* to successively input each grade. We do not know in advance how many grades will be entered, so we'll use sentinel-controlled iteration. 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 count
    Print the average
Else
    Print “No grades were entered”
```

We’re careful here to test for the possibility of *division by zero*—a *logic error* that, if undetected, would cause the program to fail or produce invalid output. The complete second refinement of the pseudocode for the class-average problem is shown in Fig. 4.9.

```
Initialize total to zero
Initialize counter to zero

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

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

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

Fig. 4.9

Class-average pseudocode algorithm with sentinel-controlled iteration.

Error-Prevention Tip 4.4

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

In Fig. 4.7 and Fig. 4.9, we included blank lines and indentation in the pseudocode to make it more readable. The blank lines separate the algorithms into their phases and set off control statements; the indentation emphasizes the bodies of the control statements.

The pseudocode algorithm in Fig. 4.9 solves the more general class-average problem. This algorithm was developed after two refinements. Sometimes more are needed.

Software Engineering Observation 4.5

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

Software Engineering Observation 4.6

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

Implementing Sentinel-Controlled Iteration

In Fig. 4.10, method `main` implements the pseudocode algorithm of Fig. 4.9. Although each grade is an integer, the averaging calculation is likely to produce a number with a *decimal point*—in other words, a real (floating-point) number. The type `int` cannot represent such a number, so this class uses type `double` to do so. You'll also see that control statements may be *stacked* on top of one another (in sequence). The `while` statement (lines 20–27) is followed in sequence by an `if…else` statement (lines 31–42). Much of the code in this program is identical to that in Fig. 4.8, so we concentrate on the new concepts.

```java
// Fig. 4.10: ClassAverage.java
// Solving the class-average problem using sentine
import java.util.Scanner; // program uses class Sc

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

      // initialization phase
      int total = 0; // initialize sum of grades
      int gradeCounter = 0; // initialize # of grad

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

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

         // prompt for input and read next grade fr
         System.out.print("Enter grade or -1 to qui
         grade = input.nextInt();
      }

      // termination phase
      // if user entered at least one grade…
      if (gradeCounter != 0) {
         // use number with decimal point to calcul
         double average = (double) total / gradeCou

         // display total and average (with two dig
         System.out.printf("%nTotal of the %d grade
            gradeCounter, total);
         System.out.printf("Class average is %.2f%n
      }
      else { // no grades were entered, so output a
41         System.out.println("No grades were entered
42      }
43   }
44 }
```

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

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

Fig. 4.10

Solving the class-average problem using sentinel-controlled iteration.

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

Line 12 initializes `gradeCounter` to `0`, because no grades have been entered yet. Remember that this program uses *sentinel-controlled iteration* to input the grades. To keep an accurate record of the number of grades entered, the program increments `gradeCounter` only when the user enters a valid grade.

Compare the logic for sentinel-controlled iteration used here with that for counter-controlled iteration in Fig. 4.8. In counter-controlled iteration, the `while` statement (lines 15–20 of Fig. 4.8) reads a value from the user for the specified number of iterations. In sentinel-controlled iteration, the program reads the first value (lines 16–17 of Fig. 4.10) *before* reaching the `while`. This value determines whether the program's flow of control should enter the `while`'s body. If the condition is `false`, the user entered the sentinel value, so the `while`'s body does not execute (i.e., no grades were entered). If, on the other hand, the condition is `true`, the body executes, and the loop adds the `grade` value to the `total` and increments the `gradeCounter` (lines 21–22). Next, lines 25–26 *in the loop body* input another value from the user. Then, program control reaches the `while`'s closing right brace (line 27), so execution continues by testing the `while`'s condition (line 20), using the most recent `grade` input by the user. The value of `grade` is always input from the user immediately before the program tests the `while` condition, so the program can determine whether the value just input is the sentinel value *before* processing that value (i.e., adds it to the `total`). If the sentinel value is input, the loop terminates, and the program does not add –1 to the `total`.

Good Programming

Practice 4.3

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

After the loop terminates, the `if…else` statement at lines 31–42 executes. The condition at line 31 determines whether any grades were input. If none were input, the `else` part (lines 40–42) of the `if…else` statement executes and displays the message `"No grades were entered"` and the program terminates.

Braces in a `while` statement

Notice the `while` statement's *block* in Fig. 4.10 (lines 20–27). Without the braces, the loop would consider its body to be only the first statement, which adds the `grade` to the `total`. The last three statements in the block would fall outside the loop body, causing the computer to interpret the code incorrectly as follows:

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

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

The preceding code would cause an *infinite loop* in the program if the user did not input the sentinel `-1` at line 17 (before the `while` statement).

Common Programming Error 4.5

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

Explicitly and Implicitly Converting Between Primitive Types

If at least one grade was entered, line 33 of Fig. 4.10 calculates the average of the grades. Recall from Fig. 4.8 that integer division yields an integer result. Even though variable `average` is declared as a `double`, if we had written the averaging calculation as

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

it would lose the fractional part of the quotient *before* the result of the division is assigned to `average`. This occurs because `total` and `gradeCounter` are *both* integers, and integer division yields an integer result.

Most averages are not whole numbers (e.g., 0, –22 and 1024). So, we calculate the class average in this example as a floating-point number. To perform a floating-point calculation with integer values, we must *temporarily* treat these values as floating-point numbers in the calculation. Java provides the **unary cast operator** to accomplish this task. Line 33 of Fig. 4.10 uses the **(double)** cast operator to create a *temporary* floating-point copy of its operand `total` (which appears to the operator's right). Using a cast operator in this manner is called **explicit conversion** or **type casting**. The value stored in `total` is still an integer.

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

Common Programming Error 4.6

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

You form a cast operator by placing parentheses around any type’s name. The operator is a **unary operator**—it takes only one operand. Java also has unary plus (`+`) and minus (`-`) operators, so you can write expressions like `-7` or `+5`. Cast operators associate from *right to left* and have the same precedence as other unary operators. 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 38 of Fig. 4.10 displays the class average. In this example, we display the class average *rounded* to the nearest hundredth. The format specifier `%.2f` in `printf`'s format control string indicates that variable `average`'s value should be displayed with two digits of precision to the right of the decimal point—indicated by `.2` in the format specifier. The three grades entered during the sample execution total 257, which yields the average 85.666666…. Method `printf` uses the precision in the format specifier to round the value to the specified number of digits. In this program, the average is rounded to the hundredths position and is displayed as `85.67`.

Floating-Point Number Precision

Floating-point numbers are not always 100% precise, but they have numerous applications. For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it may actually be 98.5999473210643. Calling this number simply 98.6 is fine for most applications involving body temperatures.

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

Owing to the imprecise nature of floating-point numbers, type `double` is preferred over type `float`, because `double` variables can represent floating-point numbers more accurately. For this reason, we primarily use type `double` throughout the book. In some applications, the precision of `float` and `double` variables will be inadequate. For precise floating-point numbers (such as those required by monetary calculations), Java provides class `BigDecimal` (package `java.math`), which we'll discuss in Chapter 8.

Common Programming Error 4.7

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