4.9 Formulating Algorithms: Counter-Controlled Iteration

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

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

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

Pseudocode Algorithm with Counter-Controlled Iteration

Let’s use pseudocode to list the actions to execute and specify the order in which they should execute. We use **counter-controlled iteration** 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 iteration is often called **definite iteration**, because the number of iterations is known *before* the loop begins executing. In this example, iteration terminates when the counter exceeds 10. This section presents a fully developed pseudocode algorithm (Fig. 4.7) and a corresponding Java program (Fig. 4.8) that implements the algorithm. In Section 4.10, we demonstrate how to use pseudocode to develop such an algorithm from scratch.

Note the references in the algorithm of Fig. 4.7 to a total and a counter. A **total** is a variable used to accumulate the sum of several values. A counter is a variable used to count—in this case, the grade counter indicates which of the 10 grades is about to be entered by the user. Variables used to store totals are normally initialized to zero before being used in a program.

Software Engineering Observation 4.2

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

```
Set total to zero
Set grade counter to one
```

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

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

Fig. 4.7

Pseudocode algorithm that uses counter-controlled iteration to solve the class-average problem.

Implementing Counter-Controlled Iteration

In Fig. 4.8, class `ClassAverage`’s `main` method implements the class-averaging algorithm described by the pseudocode in Fig. 4.7—it allows the user to enter 10 grades, then calculates and displays the average.

```
// Fig. 4.8: ClassAverage.java
// Solving the class-average problem using counter-
import java.util.Scanner; // program uses class Sca

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

      // initialization phase
      int total = 0;  // initialize sum of grades
      int gradeCounter = 1; // initialize # of gra

      // processing phase uses counter-controlled
      while (gradeCounter <= 10) { // loop 10 time
         System.out.print("Enter grade: "); // pro
         int grade = input.nextInt(); // input nex
         total = total + grade; // add grade to to
         gradeCounter = gradeCounter + 1; // incre
      }

      // termination phase
      int average = total / 10; // integer divisio

      // display total and average of grades
      System.out.printf("%nTotal of all 10 grades
      System.out.printf("Class average is %d%n", a
   }
}
```

```
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. 4.8

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

Local Variables in Method `main`

Line 8 declares and initializes `Scanner` variable `input`, which is used to read values entered by the user. Lines 11, 12, 17 and 23 declare local variables `total`, `gradeCounter`, `grade` and `average`, respectively, to be of type `int`. Variable `grade` stores the user input.

These declarations appear in the body of method `main`. Recall that variables declared in a method body are local variables and can be used only from the line of their declaration to the closing right brace of the method declaration. A local variable's declaration must appear *before* the variable is used in that method. A local variable cannot be accessed outside the method in which it's declared. Variable `grade`, declared in the body of the `while` loop, can be used only in that block.

Initialization Phase: Initializing Variables `total`

and `gradeCounter`

The assignments (in lines 11–12) initialize `total` to `0` and `gradeCounter` to `1`. These initializations occur *before* the variables are used in calculations.

Common Programming Error 4.3

Using the value of a local variable before it's initialized results in a compilation error. All local variables must be initialized before their values are used in expressions.

Error-Prevention Tip 4.3

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

Processing Phase: Reading 10 Grades from the User

Line 15 indicates that the `while` statement should continue looping (also called **iterating**) as long as `gradeCounter`'s value is less than or equal to 10. While this condition remains *true*, the `while` statement repeatedly executes the statements between the braces that delimit its body (lines 15–20).

Line 16 displays the prompt `"Enter grade: "`. Line 17 reads the grade entered by the user and assigns it to variable `grade`. Then line 18 adds the new `grade` entered by the user to the `total` and assigns the result to `total`, which replaces its previous value.

Line 19 adds `1` to `gradeCounter` to indicate that the program processed a grade and is ready to input the next grade from the user. Incrementing `gradeCounter` eventually causes it to exceed 10. Then the loop terminates, because its condition (line 15) becomes *false*.

Termination Phase: Calculating and Displaying the Class Average

When the loop terminates, line 23 performs the averaging calculation and assigns its result to the variable `average`. Line 26 uses `System.out`'s `printf` method to display the text `"Total of all 10 grades is "` followed by variable `total`'s value. Line 27 then uses `printf` to display the text `"Class average is "` followed by variable `average`’s value. When execution reaches line 28, the program terminates.

Notice that this example contains only one class, with method `main` performing all the work. In this chapter and in Chapter 3, you’ve seen examples consisting of two classes—one containing instance variables and methods that perform tasks using those variables and one containing method `main`, which creates an object of the other class and calls its methods. Occasionally, when it does not make sense to try to create a reusable class to demonstrate a concept, we’ll place the program’s statements entirely within a single class’s `main` method.

Notes on Integer Division and Truncation

The averaging calculation performed by method `main` produces an integer result. The program’s output indicates that the sum of the grade values in the sample execution is 846, which, when divided by 10, should yield the floating-point number 84.6. However, the result of the calculation `total / 10` (line 23 of Fig. 4.8) is the integer 84, because `total` and `10` are both integers. Dividing two integers results in **integer division**—any fractional part of the calculation is **truncated** (i.e., *lost*). In the next section we’ll see how to obtain a floating-point result from the averaging calculation.

Common Programming Error 4.4

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

A Note about Arithmetic Overflow

In Fig. 4.8, line 18

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

added each `grade` entered by the user to the `total`. Even this simple statement has a *potential* problem—adding the integers could result in a value that's *too large* to store in an `int` variable. This is known as **arithmetic overflow** and causes *undefined behavior*, which can lead to unintended results, as discussed at

```
http://en.wikipedia.org/wiki/
   Integer_overflow#Security_ramifications
```

Figure 2.7's `Addition` program had the same issue in line 17, which calculated the sum of two `int` values entered by the user:

```
int sum = number1 + number2; // add numbers, then sto
```

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

```
http://docs.oracle.com/javase/8/docs/api/java/lang/In
```

It's considered a good practice to ensure, *before* you perform arithmetic calculations like those in line 18 of Fig. 4.8 and line 17 of Fig. 2.7, that they will *not* overflow. The code for doing this is shown on the CERT website:

```
http://www.securecoding.cert.org
```

Just search for guideline "NUM00-J." The code uses the && (logical AND) and `||` (logical OR) operators, which are introduced in Chapter 5. In industrial-strength code, you should perform checks like these for *all* calculations.

A Deeper Look at Receiving User Input

Whenever a program receives input from the user, various problems might occur. For example, in line 17 of Fig. 4.8

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

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

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