LESSON 04 – Repetition Structures

==========

PROGRAM DESIGN WITH THE while LOOP

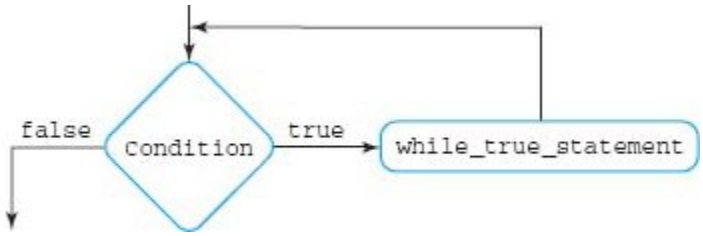
The if and if-else statements give us the ability to make choices. In this section we will see how the while statement enables us to repeat steps

Repetition

The while statement follows the pattern

```
while (condition)
  while_true_statement
```

FIGURE 3.9 Flow chart for the while loop



with the flow chart shown in Figure 3.9, where the condition evaluates to true or false, and the while_true_statement can be any C# statement including a code block. If the condition is true, C# executes the while_true_statement and goes back to check the condition again. If the condition is still true, C# executes the while_true_statement and goes back to check the condition again, and so on. This process repeats until the condition is false

For example, suppose that the variable x has the value 7 just before C# starts to execute the while statement

```
while (x < 10)
   x += 2;
```

Because 7 < 10 is true, C# executes the statement x += 2, which changes the value of x to 9. Remember, this is a while statement, so C# again checks the condition, x < 10. Because 9 < 10 is still true C# again executes, x += 2, giving x the value 11. Now checking the condition x < 10, C# finds that 11 < 10 is false, so the execution of the while statement is finished

The while statement is a type of **loop**, so called because execution keeps looping back to check the condition after every execution of the while_true_statement, which we call the **body** of the loop. The body of the loop could be a block, in which case C# executes

LESSON 04 – Repetition Structures

==========

every statement in the block while the condition is true. For example, if x had the value 5 before the loop, the while statement

```
while (x < 10) {
   Console.WriteLine('x is now {0}', x);
   x += 2;
 }
```

would output

```
x is now 5
x is now 7
x is now 9
```

The condition in a while statement may evaluate to false on the first entry to the loop, in which case C# never executes the body of the loop. For example, if x has the value 3 before executing the loop

```
while (x >= 5)
  x -= 4;
```

then the condition, 3 >= 5, is false, and the loop body, x -= 4, is never executed

Planning Programs Using Pseudocode

With the while statement we can solve more interesting problems. However, as problems get more complex, we need to plan our solutions carefully. In this section, we develop a program, Example 3.6, which uses a while loop to find and display the sum of test scores inputted by the user

Before coding, we need to plan our approach. The user will keep entering test scores, which are nonnegative. To signal to the program that no more scores are available the user can enter a negative number, which cannot be a valid score. We call this value a **sentinel.** The program checks each value the user inputs. If it is the sentinel, the program exits the while loop and outputs the total of all the scores. If it is not the sentinel the program adds the score to the total. We can use a while loop to code these repetitive steps. Informally, we could write the loop as

```
Read the first score;
while (the score is not the sentinel) {
  Add the score to the total so far;
  Read the next score;
}
Output the total of all the scores.
```

We call this informal way of writing the program **pseudocode.** Pseudocode helps us design the C# program. It is easier to understand than the very specific programming

LESSON 04 – Repetition Structures

language statements, so we can correct logical errors before they get into the actual program

We can use a flow chart in the same way as pseudocode to show the structure of a program as we develop the detailed solution. For small programs both techniques work well, but for larger programs it becomes cumbersome to manage the flow charts, making pseudocode a better choice

Pseudocode makes sense. We can execute it mentally with some test data as a further check. Suppose the scores are 56, 84, and 75, and that the user follows them with the sentinel, -1. Let us number each line and trace, line by line, how the pseudocode would execute

Pseudocode

```
1       Read the first score;
2       while ( the score is not the sentinel) {
3            Add the score to the total so far;
4            Read the next score;
5       }
6       Output the total of all the scores.
```

Trace of the pseudocode:

Line Number	Action
1	Read the first score: **56**
2	>56 is not the sentinel.
3	Add score to total: total is 56
4	Read the next score: **84**
2	84 is not the sentinel.
3	Add score to total: total is 140
4	Read the next score: **75**
2	75 is not the sentinel.
3	Add score to total: total is 215

4	Read the next score: **–1**
2	–1 is the sentinel.
6	Output the total of 215.

This trace looks correct. We should also check our pseudocode when the user has no scores to enter

Line Number	Action
1	Read the next score: **–1**
2	–1 is the sentinel.
6	Output the total of ???.

Here we get to line 6 without ever adding any scores to the total. The pseudocode always displays the total, whether or not the user entered any scores. We need to make sure that whenever we get to line 6, we have a total to display. If the user has no scores to enter, we could display zero. To set this initial value for the total, we could add a line at the beginning of the pseudocode, say line 0,

```
0.Initialize the total to zero.
```

that sets the total to zero. Then, if we do not read any scores, we will still have a total to display. Actually, we implicitly use the initial value of total in line 3 after we read the very first score. Line 3 states add the score to the total so far, but before reading any scores the total does not have a value unless we give it an initial value in line 0.

If we want to be meticulous, we might want to distinguish between the case when the total is zero because no scores were entered and when it is zero because a bunch of zero scores were entered. More broadly, we might want to keep track of the number of scores entered. Should we do these things or not? In this example, the problem does not explicitly ask to keep track of the number of scores, and for simplicity we will not add this feature now. The problem does not say what to do if no scores are available, but we assume that displaying zero is sensible, and to be sure, we check with the proposer of the problem, who agrees.

We have taken a lot of trouble to design a solution to a simple problem. Because we have been careful, we should be able to convert the pseudocode to a C# program that solves

the problem. In this chapter we use a procedural approach. When we start object-oriented programming in Chapter 5, we shall see how to design a solution using an object.

We want to abstract operations that model our understanding of the problem. Looking at the pseudocode, we identify three operations

```
Input a score from the user.
Add the score to the total.
Output the total.
```

If we define a method to implement each of these operations, we can easily translate our pseudocode into an understandable C# program.

We could use any valid identifiers, not keywords, for our method and variable names. To make our program as readable as possible, we choose meaningful names, GetScore and UpdateTotal for our method names. The GetScore method will have a String prompt as a parameter. It will return the value the user enters. We will pass the new score and the current total to the UpdateTotal method. It will return the updated total

EXAMPLE 3.6 ■ ScoreSum.cs

LESSON 04 – Repetition Structures

==========

```
/*  Uses a while loop to compute the sum of
 *  test scores.
 */

using System;
public class ScoreSum  {

     // returns a score entered by the user
  public static int GetScore(String prompt) {
    Console.Write(prompt);
    return int.Parse(Console.ReadLine());
  }

     /* Input:    The new score and the total so far
      * Output:  The updated total
      */
  public static int UpdateTotal(int newValue, int subTotal) {
                                                        // Note 1
    subTotal += newValue;
    return subTotal;
  }

     // loops until the user enters -1
  public static void Main(String [] args) {
    int total = 0;                      // Sum of scores
                                                        // Note 2
    int score = GetScore('Enter the first score, -1 to quit: ");
    while (score >= 0) {
      total = UpdateTotal(score, total);
      score =  GetScore("Enter the next score, -1 to quit: ');
    }
    Console.WriteLine("The total is {0}", total);
  }
}
```

Run

```
Enter the first score, -1 to quit:
Enter the next score, -1 to quit: 45
Enter the next score, -1 to quit: 56
Enter the next score, -1 to quit: -1
The total is 135
```

Note 1: `public static int UpdateTotal(int newValue, int subTotal){`

We choose variable names, newValue and subTotal, that remind us of the role these quantities play. Our program is easier to understand than if we had used variable names x and y for these quantities.

LESSON 04 – Repetition Structures

==========

Note 2: `int total = 0;`

We initialize total to 0, representing the sum of scores before the user has entered any scores. If the user enters the sentinel, _1, before entering any scores, the output will be 0. The C# code closely follows the steps of the pseudocode.

Having completed this program nicely, we might ask about alternative solutions in case another approach might allow us to improve our program. Instead of using a sentinel to signify the end of the input, the user might specify how many scores are available. Then we can have our loop condition check to see if all scores have been entered. If not, the body of the loop reads the next score and adds it to the total. The pseudocode for this approach is

```
Read the quantity of scores to read;
while (number of scores read so far < quantity to read) {
  Read the next score;
  Add the score to the total so far
}
Display the total;
```

Each time around the loop we need to check the condition

```
number of scores read so far < quantity to read.
```

We know how many scores the user will enter. If we have not read that many, there are more scores; otherwise, we have read them all. But how many scores have we read? Let us keep track as we read.We could include another variable, count, to hold the number of scores read so far.We initialize count to 0, and each time we read a score, we increase count by 1. The revised pseudocode is shown in Figure 3.10.

We leave the next steps in this solution process, which include tracing the execution of the pseudocode with sample data and converting it to a C# program, to the exercises. Note that to solve the problem using this approach, we had to know the quantity of test scores, so we added that to the output to make it more informative.

```
Read the quantity of scores;
while (count < quantity) {
     Read the next score;
     Add the score to the total so far;
     Increment the count of scores;
}
Display the quantity and the total;
```

FIGURE 3.10 Pseudocode for the sum of test scores problem

Think carefully when writing the condition (often called the test) for the while loop. Getting it correct can be tricky. In the preceding example we might easily have written count <= quantity, which is incorrect because when count equals quantity, then we have already read all the scores and should terminate the loop rather than ask for another score.

A Little Extra: Input Validation

In Example 3.6, we assumed that the user would enter correct data. Entering a value such as 34.5 or hat will cause the program to abort. Later we will introduce exception handling, which we can use to recover from such input errors. Even if the user enters a correct int value, that value could be 312017 or - 52, neither of which is likely to be a test score. We could have prompted the user to enter a value from 0 to 100, as in

```
Enter a score between 0 and 100
```

Even then a careless user could enter an invalid value.[6]

Style

Do	declare each variable on a separate line and use a comment to specify the use of that variable in the program.
Do	use meaningful variable names that remind us of the purpose of that variable.
Do	precede the program with a comment that describes the intent of the program, and the input and output. This comment is a good place to put the programmer's name and the date. Precede each method with a descriptive comment.
Do	include comments to explain any parts of the code that might be difficult to understand. (In our examples, we use Notes at the end of the code for this purpose.)

LESSON 04 – Repetition Structures

==========

Loop Termination

Each time the loop condition is true, C# executes the loop body. In order for the loop to terminate, something must change that causes the condition to fail. In the loop

```
while ( x < 10)
     x += 2;
```

we add 2 to x each time we execute the body. Eventually x will become greater than or equal to 10, so the condition x < 10 will fail and the loop will terminate

If we were just to display the value of x, and not increase it, then the loop might never stop. For example, if x has the value 5 when C# executes the loop

```
while (x < 10)
     Console.WriteLine('x is now {0}', x);
```

the result will be an unending sequence

```
x is now 5
x is now 5
x is now 5
...
```

and so on until someone aborts the program. (Holding the *Control* key down and pressing the *C* key will interrupt the program on Windows systems.)

The last example, repeatedly displaying x is now 5, made it easy to spot that something was very wrong. Sometimes an unending loop (often called an **infinite loop)** may have the opposite behavior, showing nothing at all to the user while it goes on computing. For example, if x has the value 5, the loop

```
while (x < 10)
  x -= 2;
```

keeps subtracting 2 from x, never terminating. The value of x is 5,3,1, -1, -3, -5, and so on. The condition, x < 10, is always true, so the loop keeps on executing, but displays nothing to the user. The programmer's first response is to blame the computer for being very slow, because nothing seems to be happening, but that eerie stillness could be a symptom of an infinite loop. If so, the user must interrupt, aborting the program

LESSON 04 – Repetition Structures

==========

Remember when writing a while statement that something must eventually cause the condition to be false

Beware of loops that never terminate. We use loops because we want repetition, but we must make sure that the repetition stops. Check each loop you write to make sure that it will terminate.