***Lab 07 – while and do-while Loops***

## Introduction (Concept Map)

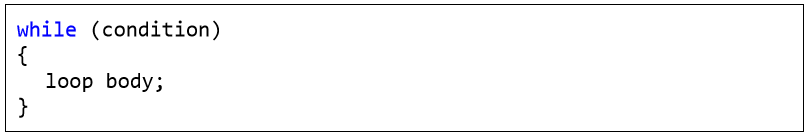
In this lab we will examine the loop programming constructs through which we can execute a code snippet repeatedly. We will discuss how to implement conditional repetitions (while and do-while loops) and how to work with for-loops.

1. Loops

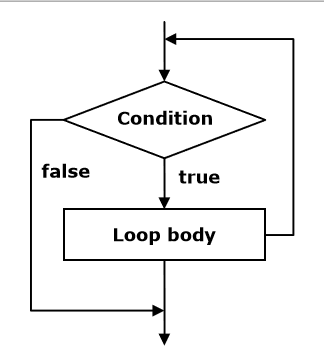
In programming often requires repeated execution of a sequence of operations. A loop is a basic programming construct that allows repeated execution of a fragment of source code. Depending on the type of the loop, the code in it is repeated a fixed number of times or repeats until a given condition is true (exists). Loops that never end are called infinite loops. Using an infinite loop is rarely needed except in cases where somewhere in the body of the loop a break operator is used to terminate its execution prematurely.

* 1. While Loop:

One of the simplest and most commonly used loops is while.

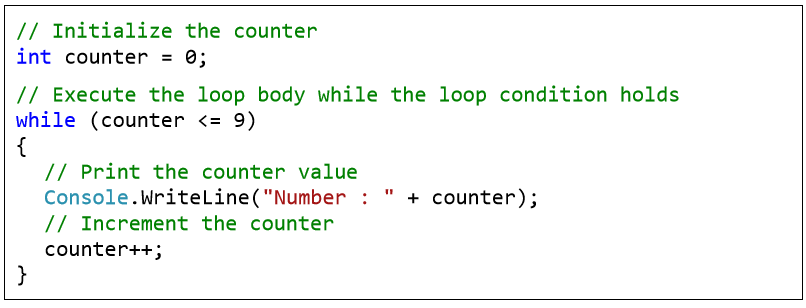


In the code above example, condition is any expression that returns a Boolean result – true or false. It determines how long the loop body will be repeated and is called the loop condition. In this example the loop body is the programming code executed at each iteration of the loop, i.e. whenever the input condition is true. The behavior of while loops can be represented by the following scheme:

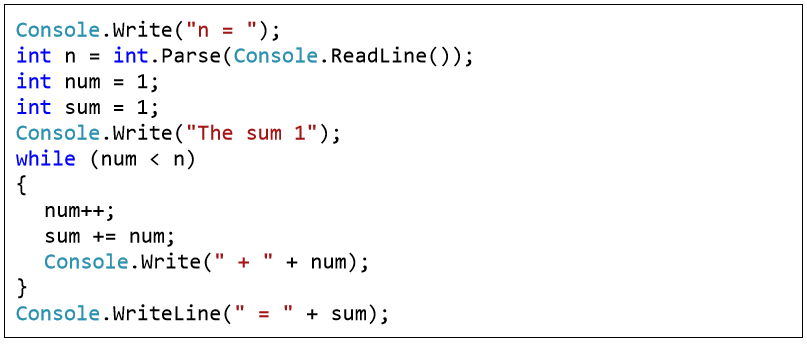


In the while loop, first of all the Boolean expression is calculated and if it is true the sequence of operations in the body of the loop is executed. Then again, the input condition is checked and if it is true again the body of the loop is executed. All this is repeated again and again until at some point the conditional expression returns value false. At this point the loop stops and the program continues to the next line, immediately after the body of the loop. The body of the while loop may not be executed even once if in the beginning the condition of the cycle returns false. If the condition of the cycle is never broken the loop will be executed indefinitely.

Let’s consider a very simple example of using the while loop. The purpose of the loop is to print on the console the numbers in the range from 0 to 9 in ascending order:



**Example 2:**



* 1. Operator "break"

The break operator is used for prematurely exiting the loop, before it has completed its execution in a natural way. When the loop reaches the break operator it is terminated and the program’s execution continues from the line immediately after the loop’s body. A loop’s termination with the break operator can only be done from its body, during an iteration of the loop. When break is executed the code in the loop’s body after it is skipped and not executed. We will demonstrate exiting from loop with break with an example.

* 1. Calculating Factorial – Example

In this example we will calculate the factorial of a number entered from the console. The calculation is performed by using an infinite while loop and the operator break. Let’s remember from the mathematics what is factorial and how it is calculated. The factorial of an integer n is a function that is calculated as a product of all integers less than or equal to n or equal to it. It is written down as n! and by definition the following formulas are valid for it: - N! = 1 \* 2 \* 3 … (n-1) \* n, for n> 1; - 2! = 1 \* 2; - 1! = 1; - 0! = 1. The product n! can be expressed by a factorial of integers less than n: - N! = (N-1)! \* N, by using the initial value of 0! = 1. In order to calculate the factorial of n we will directly use the definition:

**int n = int.Parse(Console.ReadLine());**

**// "decimal" is the biggest C# type that can hold integer values**

**decimal factorial = 1;**

**// Perform an "infinite loop"**

**while (true) {**

**if (n <= 1) {**

**break;**

**}**

**factorial \*= n; n--;**

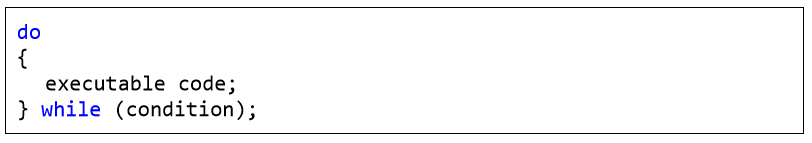
**}**

**Console.WriteLine("n! = " + factorial);**

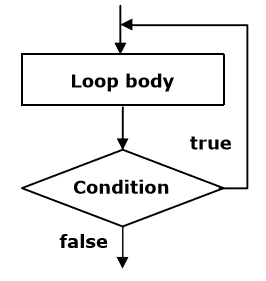
First we initialize the variable factorial with 1 and read n from the console. We construct an endless while loop by using true as a condition of the loop. We use the break operator, in order to terminate the loop, when n reaches a value less than or equal to 1. Otherwise, we multiply the current result by n and we reduce n with one unit. Practically in the first iteration of the loop the variable factorial has a value n, in the second – n\*(n-1) and so on. In the last iteration of the loop the value of factorial is the product n\*(n-1)\*(n2)\*…\*3\*2, which is the desired value of n!.

* 1. Do-While Loops

The do-while loop is similar to the while loop, but it checks the condition after each execution of its loop body. This type of loops is called loops with condition at the end (post-test loop). A do-while loop looks like this:



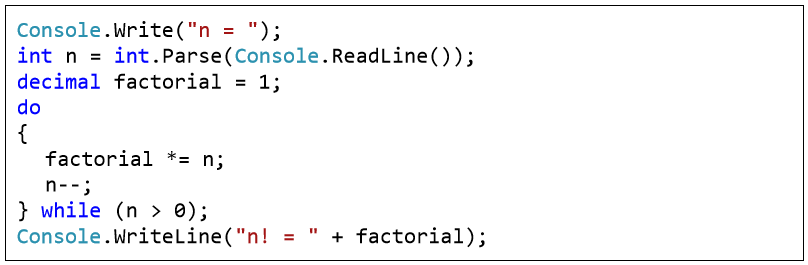
By design do-while loops are executed according to the following scheme:



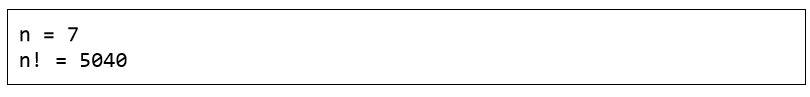
Initially the loop body is executed. Then its condition is checked. If it is true, the loop’s body is repeated, otherwise the loop ends. This logic is repeated until the condition of the loop is broken. The body of the loop is executed at least once. If the loop’s condition is constantly true, the loop never ends.

The do-while loop is used when we want to guarantee that the sequence of operations in it will be executed repeatedly and at least once in the beginning of the loop.

In this example we will again calculate the factorial of a given number n, but this time instead of an infinite while loop we will use a do-while. The logic is similar to that in the previous example:



At the beginning we start with a result of 1 and multiply consecutively the result at each iteration by n, and reduce n by one unit, until n reaches 0. This gives us the product n\*(n-1)\*…\*1. Finally, we print the result on the console. This algorithm always performs at least one multiplication and that’s why it will not work properly when n ≤ 0. Here is the result of the above example’s execution for n=7:



## Practice Lab Tasks

1. Fibonacci series ( 0,1,1,2,3,5,8…) for and while loop
2. Repeatedly print the value of the variable xValue, decreasing it by 0.5 each time, as long as xValue remains positive. (while loop)
3. Print the square roots of the first 25 odd positive integers. (while loop)