Iteration statements

Iteration statements (also known as loop statements) repeatedly execute an embedded statement while a given condition is met. The general loop concept has two integral parts – a loop condition and a loop body. The execution of the loop body is called an iteration of the loop. Depending on the loop type the condition is checked before or after the iteration is executed. If the condition is not met the loop stops execution.

The loops are divided into two types depending on where the condition is located – pre-condition and post-condition. Pre-condition loops check their condition before every iteration thus if the condition is not met no iterations will be executed. The post-condition loops on the other hand check their conditions after the iteration is executed thus at least one iteration of the loop gets executed even if the condition is never met. The iteration statements in C# are:

- while – a pre-condition loop. The syntax of the "while" statement is:

```
while(condition)
{
   // code block
}
```

This statement is the most standard example of an iteration statement. It checks the condition and then executes the code block. This operation is repeated while the condition is met.

- do-while – a post-condition loop. The syntax of the "do-while" statement is:

```
do
{
   // code block
}
while(condition)
```

This statement is a post-condition "while" statement. It executes the code block and then checks the condition. This operation is repeated while the condition is met.

- for – is a pre-condition loop. The syntax of the "for" statement is:

```
for(initialization; condition; iteration)
{
   // code block
}
```

This statement is a bit different from the last two. It contains three sections – initialization, condition and iteration section. In the initialization section we can declare variables and set initial values. The scope of the variables declared in the initialization section is limited to the loop's body – they are not visible outside of the "code block" in the syntax sample above.

The condition section holds the loop condition. While the condition is satisfied the loop will continue its execution.

The iteration section contains a statement that is executed after every single iteration of the loop.

- foreach – does not have a condition section. The syntax of the “foreach” statement is:

```
foreach(var item in collection)
{
   // code block
}
```

The “foreach” construct repeats the code block for each one of the elements in the collection. The item variable’s scope is limited by the code block of the loop. Before each iteration the item variable is assigned with the next element in the collection. The loop ends when all elements of the collection are visited.

While statement

Let’s start with a simple example. Say we need to compute the sum of the numbers from 1 to N.

```
using System;

namespace Sum
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter a natural number: ");
            int n = int.Parse(Console.ReadLine());

            int iterator = 0;
            int sum = 0;

            while (iterator <= n)
            {
                sum = sum + iterator;
                iterator = iterator + 1;
            }

            Console.WriteLine("The sum of the first {0} natural numbers is {1}", n, sum);
            Console.ReadKey(true);
        }
    }
}
```

Note how in the example above we first read N – the end of the natural numbers range we’re going to sum up. On the next line we declare an integer variable named “iterator”. The name serves as a description of the purpose of the variable. It will hold the values from 0 to N (inclusive) that will be added together. On the next line we declare the integer variable “sum” that will aggregate the sum. We initialize both “iterator” and “sum” with the value 0.

We first need to set the condition for the “while” statement. On each iteration of the loop we add the value of “iterator” to the “sum” variable, assign the result to the “sum” variable and increment the “iterator” variable. This way in “sum” we will aggregate the sum of 0 to N.

Let’s take a look at the case when the user inputs the value “3” for n. On the first iteration of the while loop both “iterator” and “sum” are 0. The “while” statement is a pre-condition one so the condition is checked – (0 <= 3) which evaluates to “true” thus we enter first iteration.

At the end of the first iteration “sum” is still 0 but “iterator” is 1. The condition is checked again – (1 <= 3) which is still “true” thus we enter second iteration. At the end of it “sum” is 1 and “iterator” is 2. The condition is still met – (1 <= 3) and we step into the third iteration.

At the end of the third iteration “sum” is 3 and “iterator” is 3. The condition is checked again – (3 <= 3) which evaluates to “true”. We’re execute our last iteration.

At the end of the forth iteration “sum” is 6 and “iterator” is 4. The condition is no longer met – (6 <= 3) thus we break the loop.

At the next line we output the sum to the standard output.

This is a very simple aggregation example. Note that there are short syntaxes for aggregation and incrementation. For example the “while” statement can look like this:

```
            while (iterator <= n)
            {
                sum += iterator;
                iterator++;
            }
```

Our next example will still be aggregation related. This time we need the product of the natural numbers from 0 to N. This is practically the same problem. The only difference is in variable initialization and the loop’s body.

We can’t initialize the “iterator” with the value 0 because a product with a single 0 factor will be zero itself.

Note that similarly to the “+=” operator in the loop’s body we’re using the “*=” operator. This is equivalent to the statement “sum = sum * iterator”. This operator aggregates the right operand in the left operand by multiplying the two values and assigning the result in the left operand.

At the end we output the result to the standart output.

```
using System;

namespace Product
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter a natural number: ");
            int n = int.Parse(Console.ReadLine());

            int iterator = 1;
            int sum = 1;

            while (iterator <= n)
            {
                sum *= iterator;
                iterator++;
            }

            Console.WriteLine("The product of 0 to {0} is {1}", n, sum);
            Console.ReadKey(true);
        }
    }
}
```

Do statement

Let’s start with a simple example of the do-while post-condition iteration statement. In this example we’ll create a program that will talk back to us.

In our example we will hard code fixed conversation lines with appropriate answers. Whenever the user inputs one of them we will return the corresponding fixed answer. The do-while statement is used in cases like this one where the first iteration should be executed even if the condition is not met. See below that we initialize the “phrase” variable with a value that doesn’t satisfy the condition, but being a post-condition loop do-while will not check the condition until the end of the first iteration so the initialization value will not be relevant.

In our case we need to urge the user to start talking to our program so we declare a string variable where we will record his phrases. On the next two lines we output a quick salute and we urge him to write back to our program. We read a line from the standard input and assign it to the “phrase” variable.

On the next few lines we have “if” statements where we check the user input. If it matches one of our hard coded phrases our program outputs to the standard output a corresponding answer. If no matches are found the “Computer” says that he didn’t understand the user.

The loop is broken when the user inputs “Bye!”.

At the end of the code you’ll notice a Console.Clear() call. This will clear all text in the console.

```
using System;

namespace Conversation
{
    class Program
    {
        static void Main(string[] args)
        {
            string phrase = "Bye!";

            Console.WriteLine("Computer says: Hi!");

            do
            {
                Console.Write("You say: ");
                phrase = Console.ReadLine();

                if (phrase == "What's your name?")
                {
                    Console.WriteLine("Computer says: My name is Computer.");
                }
                else if (phrase == "What's 2 + 2?")
                {
                    Console.WriteLine("Computer says: 4!");
                }
                else
                {
                    Console.WriteLine("Computer says: I didn't understand this.");
                }
            }
            while (phrase != "Bye!");

            Console.Clear();
            Console.WriteLine("Hope you enjoyed our conversation!");

            Console.ReadKey(true);
        }
    }
}
```

For statements

We continue with the “for” iteration statement. The example we’re going to consider is the interest on yearly basis of a deposit.

Say we have a deposit with a 3% yearly interest. Our deposit is 5067 currency units. Our program should compute our deposit’s balance after a five years period when. We know that the deposit is incremented with the interest on yearly basis and the conditions of the deposit do not change in time.

In order to do this we need to calculate the interest for each one of the five years. For that we declare variables that will hold the interest percentage, the initial deposit amount and the number of years that we plan to keep our cash in the bank. At the end of the program the “final_amount” variable will hold the aggregated amount after the five year deposit.

We initialize the “final_amount” with the amount of the deposit. In the “for” loop we declare the variable “i” which will be our iterator that will count the iterations that the loop must execute. In our case they are 5 – from 0 to 4 inclusive. That is why the condition is (i < years).

On each iteration the “final_amount” is incremented with the current amount multiplied by the interest. On the first iteration (the beginning of the first year of our deposit) the “final_amount” is equal to the “deposit”. At the end of the first iteration our “final_amount” is incremented with the interest that’s due at the end of the first year of our deposit. At the beginning of the second iteration (the beginning of the second year of our deposit) the “final_amount” equals the “deposit” plus the interest for the first year. This is the amount we’re going to deposit for our second year. At the end of the second iteration (the end of the second year of the deposit) our amount bears interest based the amount at the beginning of the year.

This is repeated five times (once for each year of the deposit). At the end of the program we output the final amount of the deposit after the five years.

```
using System;

namespace Deposit
{
    class Program
    {
        static void Main(string[] args)
        {
            double interest = 0.03;
            double deposit = 5067;
            int years = 5;

            double final_amount = deposit;

            for (int i = 0; i < years; i++)
                final_amount += final_amount * interest;

            Console.WriteLine("The final amount of the deposit is {0:F2}", final_amount);
            Console.ReadKey(true);
        }
    }
}
```

Break and Continue

There are two operators that manage the execution of iteration statements. One is for explicitly breaking the loop the other is for skipping iterations. Let’s see their behavior by example.

We need to sum up all odd natural numbers from 0 to N:

```
using System;

namespace SumOddNumbers
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("We are going to sum up the odd numbers from 0 to ...");
            int n = int.Parse(Console.ReadLine());
            int sum = 0;

            for (int i = 0; i <= n; i++)
            {
                if ((i % 2) == 0)
                {
                    continue;
                }

                sum += i;
            }

            Console.WriteLine("The sum of all odd numbers from 0 to {0} is {1}", n, sum);
            Console.ReadKey(true);
        }
    }
}
```

Similarly to the previous problems here we declare “n” as the end of the range and read it from the standard input. The “sum” variable holds the aggregated sum. Let’s take a look at the loop’s body. Here we check if the 2 divides the iterator “i” without remainder. If so then “i” is even and should not be aggregated in “sum”. The “continue” statement skips the rest of the loop’s code block and continues with the next iteration.

The “for” loop contains an iteration statement section. It will be executed since it is outside of the code block of the loop. The “wile” and “do-while” loops work in a simpler manner since they don’t have such section.

Now that “continue” is clear let’s have an example of the “break” statement. Say we need to compute the sum of all even numbers in the closed range 0 to N. In the example below you can see that once again we declare “n” and “sum”. Since we’re not using the “for” loop we need to declare our iterator “i” outside the loop in order to use it globally in our Main function.

Note that the condition of our “while” loop is a constant value – “true”. This is called endless loop because the condition always evaluates to “true” thus the loop will never stop. Sometimes we need to place the condition for breaking the loop inside its code block. That’s what we’re about to do here.

If the iterator is even we aggregate it in the “sum” variable. Right after that we increment the iterator and check whether it has exceeded the end of the range. If so we use the “break” statement to stop the loop explicitly.

```
using System;

namespace SumEvenNumbers
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("We are going to sum up the odd numbers from 0 to ...");
            int n = int.Parse(Console.ReadLine());
            int sum = 0;

            int i = 0;
            while(true)
            {
                if ((i % 2) == 0)
                {
                    sum += i;
                }

                i++;
                if (i > n)
                {
                    break;
                }
            }

            Console.WriteLine("The sum of all odd numbers from 0 to {0} is {1}", n, sum);
            Console.ReadKey(true);
        }
    }
}
```

The “break” and “continue” statement are applicable in all iteration statements. They give flexibility and are very useful when implementing complex algorithms.

Now say we need to output all the permutations of the natural numbers 1, 2 and 3:

```
using System;
namespace Permutations
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 1; i <= 3; i++)
            {
                for (int j = 1; j <= 3; j++)
                {
                    Console.Write("({0}, {1})\t", i, j);
                }
            }
            Console.ReadKey(true);
        }
    }
}
```

In order to do that we need to nest two loops. We need the permutations of two elements. The first loop with iterate through the available values for the first element – 1,2 and 3. For each of these values the second loop will iteration through the available values for the second element – 1,2 and 3.

As a result while the first loop's iterator "i" has value 1 the second loop's iterator "j" will iterate and sequentially have the values 1,2 and 3. In each iteration of the second loop we will output the value of the first iterator "i" and the second iterator "j" thus the result will be (1,1) (1,2) and (1,3). Once "j" has value 4 we exit the second loop and go back to the first where "i" has now value 2 and we enter the second loop once again. The iterator "j" is declared again and initialized with 1 (note that "j" lives only inside the second loop and can't be used outside of it).

The same operation is repeated but this time "i" is 2 thus the result is (2, 1) (2, 2) (2, 3). This is repeated while the first loop's condition is still met (i <= 3).

Now that this example is clear let's try something a bit harder. Let's output a 5 symbol high triangle formed of asterisks:

```
using System;

namespace AsteriskTriangle
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 5; i++)
            {
                for (int j = 0; j < i; j++)
                {
                    Console.Write("* ");
                }
                Console.WriteLine();
            }

            Console.ReadKey(true);
        }
    }
}
```

Let's picture the console as a grid. In order to draw a triangle we need to output the asterisk symbol in some of the grid's cells. To do that we need to nest two loops – the first one will handle the rows of the grid and the second the columns. We need to write a single symbol to the first line, two on the second and so on until we reach the fifth line where we draw five symbols. It's obvious that on line n we draw n symbols (one on the first, two on the second and so on).

Since the iterator "i" handles the line index we can easily iterate from 0 to "i" and output "i" asterisk symbols. This is handled by the second "for" loop. Its iterator "j" is initialized with 0 and incremented at each iteration until it exceeds the value of "i". This means that the second loop will iterate "i" times outputting "i" asterisk symbols. As a result we output our triangle.