Looping Based on a Logical Expression

Expression-Based Loops

You can execute the same statement multiple times in your program by using a *loop*. There are several different kinds of loops in C#. In this lession, you're going to learn about the *while loop*, which is created with the `while` keyword. A `while` loop in C# looks identical to an `if` statement, except that instead of executing the contents of the associated block just once, the `while` loop will execute them again and again as long as the logical expression associated with the loop evaluates to `true` .

In previous lessons, you were asked to write some guessing games in which the user could type in a guess, and your program would let them know if they were correct, or not. Unfortunately, the user only got to make one guess, and then the program ended. If they weren't correct, they didn't get another chance. Using a `while` loop, you can easily create a game that continues until the user guesses correctly:

```
int numberToGuess = new Random().Next(1,101); // a number from 1 to 100
int currentGuess = 0; // start with an incorrect guess
while (currentGuess != numberToGuess)
{
    Console.WriteLine("Guess the number (1 to 100): ");
    currentGuess = int.Parse(Console.ReadLine());
    if (currentGuess < numberToGuess)
    {
        Console.WriteLine("You guessed too low! Try again!");
    }
    if (currentGuess > numberToGuess)
    {
        Console.WriteLine("You guessed too high! Try again!");
    }
}
Console.WriteLine("You got it! Good job!");
```

Using `int.Parse(Console.ReadLine())` can easily crash the program if the user enters a non-integer value (try it!). You'll learn how to validate user input and avoid such issues later in this tutorial.

You can think of a `while` loop as being just like an `if` statement that keeps running, again and again, until the expression is evaluated as `false` . Note that the expression is only evaluated when the end of the loop block is reached. It's possible for the expression to evaluate to false at some point during the block's execution, but for the loop to continue as long as the expression is true again the next time it is evaluated.

> **Tip** {.tip .newLanguage}
> When a loop is based on the value of a particular variable, this variable is called the *loop control variable*. It's often critical that you remember to update the value of the loop control variable within the loop, or risk an infinite loop (see below).

Exiting a Loop

There are several ways you can exit a loop in C#. The first one is for the loop conditional expression to evaluate to `false`. When that happens, the loop will skip its block of code and execution will continue on the next statement after the loop code block.

You can also exit a loop by using the `break` keyword. You learned about this previously in the context of `switch` statements. The `break` keyword will break out of the current loop or switch statement and continue execution on the next statement in the program. `break` can be a much cleaner and more direct way to exit a `while` loop than trying to set the loop condition to `false` and avoiding execution of the rest of the code block for the current loop. Sometimes you want execution to get out of the loop *now*, and `break` is the best way to do that.

You can also use the `return` keyword to exit a loop. This will not only exit the loop, but will return execution from the currently executing method. You'll learn more about methods, soon.

You can also exit a loop by throwing an *exception*. You'll learn more about those later in this tutorial, as well.

Infinite Loops

Many programs, especially games, will keep running, waiting for user input, until the user takes some action to exit them. If you suspect that your console program has entered an infinite loop, you can force it to end by pressing [Ctrl]+c.

> **Tip** {.tip .newLanguage}
> An infinite loop is a loop that never ends. Often, these are a programming bug caused by broken logic that includes a loop conditional expression that never evaluates to `false`. Sometimes, an infinite loop is intentional.

You can modify the guessing game shown above to use an infinite loop and the `break` keyword (and at the same time, making it a bit shorter):

```
int numberToGuess = new Random().Next(1,101); // a number from 1 to 100
while (true) // this sets up an infinite loop, since true will always evaluate to true
{
    Console.WriteLine("Guess the number (1 to 100): ");
    int currentGuess = int.Parse(Console.ReadLine());
    if (currentGuess==numberToGuess) break;
    if (currentGuess < numberToGuess)
    {
        Console.WriteLine("You guessed too low! Try again!");
    }
    if (currentGuess > numberToGuess)
    {
        Console.WriteLine("You guessed too high! Try again!");
```

```
    }
}
Console.WriteLine("You got it! Good job!");
```

Designing Conditional Loops

It's often useful when approaching a complex problem that will require a conditional loop to start out with an `if` statement, and then once that's working, modify it to be a `while` statement. Start with a simple case, representing one pass through the loop, and implement this using an `if` statement. When the logic is working correctly, you can expand it from a one-time conditional to a loop by simply substituting `while` for `if` . This is especially common if you're testing individual cases as you write your software. The first case might not require any conditional logic at all. The second case might require an `if` . And later cases might require the `if` to be converted into a `while` . Keep this relationship between `if` and `while` in mind as you write your C# programs.

Next Steps

The following program uses an `if` statement to determine whether a given number is divisible by 2. Modify the program to change the two outer `if` statements into `while` loops so that the program lists all of the factors of the number the user supplies. You will need to ensure you update the values of both the `number` and `factor` variables within the loop(s) to avoid an infinite loop condition.

Note that the `%` operator used below is the modulus operator. It returns the remainder of an integer division operation. Examples:

```
3 % 2 // 1
10 % 2 // 0
```

Run the program as-is before modifying it. It should display "2" if you enter an even number, or nothing otherwise.

```
public static void Main()
{
    Console.WriteLine("Enter a number:");
    int number = int.Parse(Console.ReadLine());
    Console.Write("Factors: ");
    if (number > 1) // convert this to while
    {
        int candidateFactor = 2;
        if (candidateFactor <= number) // convert this to while
        {
            if (number % candidateFactor == 0) // found a factor
            {
                Console.Write(candidateFactor);
                // divide number by the factor you found and assign this back to number
                // print a comma if number is still greater than 1
```

```
            }
            // don't forget to increment factor!
        }
    }
    Console.WriteLine();
}
```