Looping a Known Number of Times

For Loops

When you know the number of times you want to execute the code inside of your loop, or it is easily calculated, the `for` loop in C# is often the best available option. It looks and works like the `while` loop you used in the previous lesson, however, it has some additional features that you'll use in this lesson. For the simple case of looping over an operation a known-in-advance number of times, the `for` loop is the preferred loop.

Using a `for` loop, you can easily write a loop that will print a list of numbers to the screen:

```
public static void Main()
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(i);
    }
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
        }
    }
}
```

This loop will write the numbers 0 through 9 to the console, executing a total of 10 times. Unlike the single, boolean expression in the `while` loops you've seen, there are three *expressions* in the first line of a `for` loop. The first expression in the example loop declares an integer variable, `i`. The `int i = 0;` creates a new variable with an initial value of 0, the first time through the loop, the value of that variable, `i`, is 0. The last time the loop executes, the value of `i` is 9, because the conditional expression `i < 10` says to continue only if the value of `i` is less than 10. In a moment, you'll see how and why the value of `i` is changing.

Tip {.tip .newLanguage }
The *loop control variable* used most often is an integer variable with the name `i`. This is a common convention used across languages in programming.

Starting From Different Values

In the previous example, the first value in the loop is 0. If you care about the number of times a loop executes, but not about the value, that might be fine. If you do care about the value of the variable, you can start it wherever you like. If you adjust the `for` loop to count like most people do, you will want to have it start at one. And when you change it, you will have code like this:

```
public static void Main()
{
    for (int i = 1; i < 10; i++)
    {
        Console.WriteLine(i);
    }
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        for (int i = 1; i < 10; i++)
        {
            Console.WriteLine(i);
        }
    }
}
```

This code will count from 1 to 9 instead of starting at 0. In fact, you can choose many other values as the starting number for that loop (even negative numbers). Go ahead and give it a try.

You can use variables to set the initial value of `i` or in your *conditional*, so your loop becomes dynamic:

```
public static void Main()
{
    int startingNumber = 5; // change to whatever value you want to start from
    int endingNumber = 10; // change to whatever number you want to be the last displayed

    for (int i = startingNumber; i <= endingNumber; i++)
    {
        Console.WriteLine(i);
    }
}
```

```
using System;

public class Program
```

```
{
    public static void Main()
    {
        int startingNumber = 5; // change to whatever value you want to start from
        int endingNumber = 10; // change to whatever number you want to be the last
displayed

        for (int i = startingNumber; i <= endingNumber; i++)
        {
            Console.WriteLine(i);
        }
    }
}
```

These first two expressions you have been learning about in the `for` loop are called the *initialization* and the *condition*. The first initializes the starting point, and the second is the condition that must be true in order to continue looping.

Tip {.tip .newLanguage }
By convention, for loops use integer loop control variables and equality checks against this variable in the condition. You can perform other operations in these expressions, but it's highly discouraged.

Counting Up By Different Increments

Sometimes you will want to increment your for loop by something other than 1 each time. The third expression in the `for` loop declaration, the *afterthought*, executes after each time through the loop. In the examples you've seen so far, the loops have used `i++` to increase the value of `i` by 1 after each time through the loop. By convention, it simply updates the loop control variable. It doesn't need to be followed by a `;` because it's already at the end of the `for` construct. The `++` operator is shorthand for "increment by 1". The statement `i++` is equivalent to `i = i + 1`.

The following example shows how you can create a loop that includes only odd numbers and prints them to the screen:

```
Console.WriteLine("Odd Numbers from 1-49:");
for (int i = 1; i < 50; i+=2)
{
    Console.WriteLine(i);
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Odd Numbers from 1-49:");
```

```
        for (int i = 1; i < 50; i+=2)
        {
            Console.WriteLine(i);
        }
    }
}
```

In the example, instead of using the `i++` operation, a new operator, `+=`, is shown. This operator is shorthand for "add a value to this variable and assign the sum back to the variable itself", So `i+=2` is equivalent to `i = i + 2`. And, of course, `i+=1` would be equivalent to `i++`.

Counting Down

Although most `for` loops start from a known value and increment by one until they reach a maximum, there's nothing in the syntax that requires this. You can use `for` loops to count down as easily as to count up. For example, the following loop will count down from 10:

```
Console.WriteLine("Countdown started...");
for (int i=10; i > 0; i--)
{
    Console.WriteLine(i);
}
Console.WriteLine("LIFTOFF!");
```

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Countdown started...");
        for (int i=10; i > 0; i--)
        {
            Console.WriteLine(i);
        }
        Console.WriteLine("LIFTOFF!");
    }
}
```

In the above sample, you can see the `--` operator at work. This is analagous to the `++` operator, except instead of incrementing the variable, it's decrementing it. `i--` is equivalent to `i = i - 1`. And, of course, the `-=` operator, which is analagous to `+=`, exists as well, so `i-=1` would also be a valid way to decrement a number by 1.

Nested Loops

It's not unusual to need to have loops within loops. For example, to display a table of values that includes multiple rows and columns, a typical approach is to start looping through the rows, and within each row, loop through the columns, displaying each value. When creating basic `for` loops using this approach, the loop control variables are conventionally named, in sequence, `i`, `j`, `k`. If you need more than three levels of nested `for` loops in your program, it's probably worth reconsidering your design or extracting some of the logic out into another method.

The following example prints out a table showing the results of multiplying the numbers 1 through 9 by one another:

```
public static void Main()
{
    Console.WriteLine("Multiplication Table:");
    Console.WriteLine("     1  2  3  4  5  6  7  8  9");
    for (int i = 1; i < 10; i++)
    {
        Console.Write($" {i} ");
        for (int j = 1; j < 10; j++)
        {
            string product = (i * j).ToString();
            Console.Write(product.PadLeft(3));
        }
        Console.WriteLine();
    }
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Multiplication Table:");
        Console.WriteLine("     1  2  3  4  5  6  7  8  9");
        for (int i = 1; i < 10; i++)
        {
            Console.Write($" {i} ");
            for (int j = 1; j < 10; j++)
            {
                string product = (i * j).ToString();
                Console.Write(product.PadLeft(3));
            }
            Console.WriteLine();
        }
    }
}
```

Althought many developers would find the above example acceptable given its use of standard loop control variable names `i` and `j`, it's often worth using more descriptive names when possible. In this case, since `i` represents the current row, it could be renamed `rowIndex`. Likewise, `j` represents the current column, and so could be renamed `columnIndex` or `colIndex`. Take care to choose good names for your variables, as this greatly improves the readability and maintainability of your programs.

Note: The above program uses a couple of new `string` operations you may not be familiar with. First, `ToString()` can be called on any object to get string representation of it. In this case, the program is using it to convert an integer value into a string version of the same value (so, `5` becomes `"5"`). Second, the `PadLeft(3)` method is used to add spaces to the beginning of a string until it reaches a certain length (in this case, 3). It's useful for working with fixed-width tables like the one this program is producing (which lines up nicely as a grid in a console window, but isn't as pretty in a browser window).