CS109
Chapter 6
Loops

Last time we looked at how to use if-then statements to control the flow of a program. In this section we will look at different ways to repeat blocks of statements. Such repetitions are called loops and are a powerful way to perform some task over and over again that would typically be too much work to do by hand. There are several ways to construct loops. We will examine the while and for loop constructs here.

**While Loop**

The while loop allows you to direct the computer to execute the statement in the body of the loop as long as the expression within the parentheses evaluates to true. The format for the while loop is:

```
While (boolean_expression)
    statement1;
    ...
    statement N;
End While
```

As long as the Boolean expression evaluates to true, statements 1 through N will continue to be executed. Generally one of these statements will eventually make the Boolean expression become false, and the loop will exit.

In terms of a flowchart, the while loop behaves as follows:
An alternate way to write a while loop is as a Do-While loop. The syntax is:

\[
\begin{align*}
\text{Do While (boolean\_condition)} \\
& \quad \text{Statement 1} \\
& \quad \ldots \\
& \quad \text{Statement N} \\
\text{Loop}
\end{align*}
\]

We’ll revisit the do while loop later. Normally I will use the regular While Loop format.

Here is an example of a while loop that prints the numbers from 1 to 10:

```csharp
Dim i As Integer = 0
While (i <= 10)
    Console.WriteLine(i)
    i = i + 1
End While
```

If we wanted to print out 1,000,000 numbers we could easily do so by changing the loop! Without the while loop, we would need 1,000,000 different WriteLine statements, certainly an unpleasant task for a programmer. Similarly, you might recall an earlier example where we read data from a file with multiple ReadLine statements. If there were hundreds of items in the file, it would be much better to read everything using a loop.

There are two types of while loops that we can construct. The first is a count-based loop, like the one we just used above. The loop continues incrementing a counter each time, until the counter reaches some maximum number. The second is an event-based loop, where the loop continues indefinitely until some event happens that makes the loop stop. Here is an example of an event-based loop:

```csharp
Dim i As Integer = 0
Dim sum As Integer = 0
While (i <> -9999)
    i = CInt(InputBox("Enter an integer, -9999 to stop"))
    If (i <> -9999) Then
        sum = sum + i
    End If
End While
Console.WriteLine("The total is " & sum)
```

This loop will input a number and add it to sum as long as the number entered is not –9999. Once –9999 is entered, the loop will exit and the sum will be printed. This is an event-based loop because the loop does not terminate until some event happens – in this case, the special value of –9999 is entered. This value is called a sentinel because it signals the end of input. Note that it becomes possible to enter the sentinel value as
data, so we have to make sure we check for this if we don’t want it to be added to the sum.

What is wrong with the following code? Hint: It results in what is called an infinite loop.

```vbnet
Dim x as Integer = 1
Dim y as Integer = 1

While (x<=10)
    Console.WriteLine(y)
    y=y+1
End While
```

Exercise: Write a program that outputs all 99 stanzas of the “99 bottles of beer on the wall” song.

For example, the song will initially start as:

99 bottles of beer on the wall, 99 bottles of beer,
take one down, pass it around,
98 bottles of beer on the wall.

Write a loop so that we can output the entire song, starting from ninety-nine and counting down to zero.

It is also possible to put a loop inside a loop. You really have no restrictions about the type of statements that can go in a loop! This type of construct is called a nested loop. The inner loop must be fully contained inside the outer loop:

```
While (bool1)
    While (bool2)
        End While
    End While
```
Example: What is the output of this code?

```csharp
While i < 6
    j = 0
    While j < i
        Console.Write("*")
        j = j + 1
    End While
    Console.WriteLine()
    i = i + 1
End While
```

Nested loops are quite common, especially for processing tables of data.

**Loop Until**

It turns out that we can do all of the looping we need with the while loop. However, there are a number of other looping constructs make it easier to write certain kinds of loops than others. Consider the loop-until loop, which has the following format:

```csharp
Do
    statement1;
    statement2;
    ...
    statement N;
Loop Until (Boolean_condition);
```

The Loop Until executes all of the statements until the Boolean condition is true; that is, the loop continues while the Boolean condition is false. In the Loop Until, the computer **always** executes the body of the loop at least once before it checks the Boolean condition. In the while-do loop, the Boolean condition is checked **first**. If it is false, then the loop’s body is never executed.

For example, we could rewrite the following While Loop as a Loop Until:

```csharp
While (Boolean_Condition)
    Statements
End While
```

Into:

```csharp
If (Boolean_condition)
    Do
        Statements
    Loop Until (Not(Boolean_condition))
End If
```
As an example, let’s convert the while loop we wrote to input numbers into a loop-until.

Dim i As Integer = 0
Dim sum As Integer = 0
Do
    i = CInt(InputBox("Enter an integer, -9999 to stop"))
    If (i <> -9999) Then
        sum = sum + i
    End If
Loop Until (i = -9999)
Console.WriteLine("The total is " & sum)

Note that in the while loop we continue while i<>-9999. In this case, we write the loop as continuing until i=-9999, which is the opposite of i<>-9999.

Another place where a do-until loop is useful is to print menus and check for valid input:

Dim i As Integer
Do
    i = CInt(InputBox("Enter 1 for task 1, and 2 for task 2", "Main Menu"))
Loop Until ((i = 1) Or (i = 2))

This loop will continue as long as the user types in something that is neither ‘1’ nor ‘2’.

VB.NET allows for the use of either the While keyword or the Until keyword at the top or the bottom of a loop. As we have seen above, when using a While we continue to loop as long as the Boolean condition is true. When using a Until we continue to loop as long as the Boolean condition is false.

We will only use While at the top and Until at the bottom as this is a fairly standard convention in Visual Basic.

**The For Loop**

The for loop is a compact way to initialize variables, execute the body of a loop, and change the contents of variables. It is typically used when we know how many times we want the loop to execute – i.e. a counter controlled loop. The syntax is shown below:
The for loop above can be described in terms of an equivalent while loop:

```
i = m
While (i <= n)
    Statement(s)
    i = i + 1
End While
```

The basic for loop counts over the loop control variable, i, starting at value m and ending at value n.

Here is our loop to print ten numbers as a for loop:

```csharp
Dim i As Integer
For i = 1 To 10
    Console.WriteLine(i)
Next
```

Suppose the Anchorage population is 300,000 in the year 2002 and is growing at the rate of 3 percent per year. The following for loop shows the population each year until 2006:

```csharp
Dim pop As Integer = 300000
Dim yr As Integer
For yr = 2002 to 2006
    Console.WriteLine(yr & "  pop=" & pop)
    Pop += 0.03 * pop
Next
```

Optionally, we can add the keyword `Step` followed by a value at the end of the For line. This specifies the value that the index variable should be changed each loop iteration. If this is left off, we have seen that the loop is incremented by 1. Here is the new format:

```
For i = m to n Step s
    Statement(s)
Next
```

Instead of setting `i = i+1` at the end of the for loop, instead this sets `i=i+s` to the end of the loop. We can use this construct to count backwards or forwards in amounts not equal to 1.

The following prints out the numbers from 10 down to 1:

```csharp
Dim i As Integer
For i = 10 To 1 Step -1
    Console.WriteLine(i)
Next
```
The following shows one way to reverse a string:

```vbnet
Dim sOriginal, sReverse As String
Dim j As Integer
sOriginal = "Kenrick"
sReverse = ""
For j = sOriginal.Length() - 1 To 0 Step -1
    sReverse &= sOriginal.Substring(j, 1)
Next
Console.WriteLine(sReverse)
```

This sums the odd integers between 1 and 10:

```vbnet
Dim i As Integer
Dim s As Integer = 0
For i = 1 To 10 Step 2
    s += i
Next
Console.WriteLine(s)
```

Note that the program above outputs 25; this is 1+3+5+7+9. However, on the last iteration, i set to 11. The loop stops since i is greater than 10; this is pointed out since i is not equal to 10.

For any for loop of the form:

```
For i = m To n Step s
```

The loop will be executed exactly once if m equals n no matter what value s has. The loop will not be executed at all if m is greater than n and s is positive, or if m is less than n and s is negative.

Each for must also be paired with a Next.

Just as we constructed nested loops using the While statement, we can also make nested loops using for statements. Just as with the while loops, nested for loops must be completely contained inside the outer loop:
Here is an example to create a multiplication table:

```vba
For j = 1 To 3
    row = ""
    For k = 1 To 3
        entry = j & " x " & k & " = " & (j * k)
        row &= entry & " 
    Next
    lstTable.Items.Add(row)
Next
```

The resulting output is:
In-Class Exercise: Write a program that inputs from the user how many numbers she would like to enter. The program should then input that many numbers and computes the average of the numbers. All input should be via InputBox.

In-Class Exercise: Write a program that finds and prints all of the prime numbers between 3 and 100. A prime number is a number such that one and itself are the only numbers that evenly divide it (e.g., 3,5,7,11,13,17, …)

Here is some pseudocode:

```
Loop from i=3 to 100
    Set flag = IsAPrime
    Loop from j=2 to i-1 (could we loop to a smaller number?)
        If i divided by j has no remainder
            Then j evenly divides into i and i is not a prime number
            Set flag = IsNotAPrime
    If flag still equals IsAPrime then
        Write i “is a prime number”.
```