

## Practice Lab 4 – Loops

In this lab, we will understand the different uses of the three types of loop logic structures, and using counters (incrementing by adding a constant value) within them, which decide when the loop should terminate. We shall also learn that relational and logical operators are used in conditional statements to control the flow of execution of the code, and they evaluate to a Boolean type.

### Initial Task - Before cloning the repository

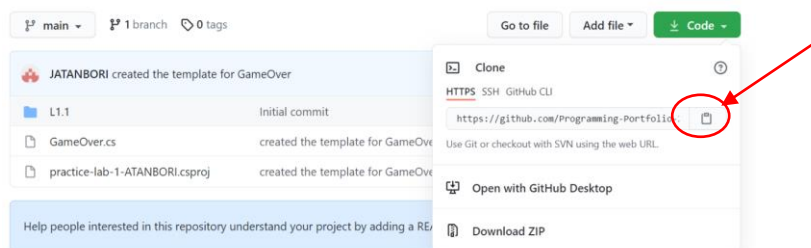
To complete the tasks for this Practice Lab, you should have created a GitHub account and joined the GitHub classroom specified above. We will use GitHub from within Visual Studio Code (VSCode), which will enable us to use some of your favourite parts of GitHub.

### Cloning the repository

1. Create a Local folder to store these practice exercises.
2. Ensure all folders are closed in VSCode (File -> Close Folder)
3. From the activity, bar click the explorer icon (ctrl + shift + E)
4. Click to view your repository, then copy the link to your repository to the clipboard.

<https://classroom.github.com/a/qgKOSv6b>

5. Click the Clone Repository button from the source control panel in VSCode.
6. When prompted to enter the repository URL, paste the data in the clipboard from step 4.



7. Select destination local copy folder and press the Select Repository Location button.

## Introduction

In this lab, you will learn to repeat parts of your code using while, do-while and for-loops. You will also learn how to use counters within loop blocks. We will control loops using Boolean values, which is somewhat similar to Booleans used in decisions. As a reminder, we assign a Boolean a value of true or false. Often we use Boolean values that are the result of relations or logical operators. Let's recap on the relational and logical operators.

The table below explains the relational operators.

Operator	Description	Examples	Result
>	<b>Greater Than</b> - Evaluates as true if the left hand operand is greater than the right hand operand. Otherwise evaluates to false.	10 > 5	True
		3 > 7	False
		4 > 4	False
<	<b>Less Than</b> - Evaluates as true if the left hand operand is less than the right hand operand. Otherwise evaluates to false.	10 < 5	False
		3 < 7	True
		4 < 4	False
>=	<b>Greater Than or Equal To</b> - Evaluates as true if the left hand operand is greater than or equal to the right hand operand. Otherwise evaluates to false.	10 >= 5	True
		3 >= 7	False
		4 >= 4	True
<=	<b>Less Than or Equal To</b> - Evaluates as true if the left hand operand is less than or equal to the right hand operand. Otherwise evaluates to false.	10 <= 5	False
		3 <= 7	True
		4 <= 4	True
==	<b>Equal To</b> - Evaluates as true if the left hand operand is equal to the right hand operand. Otherwise evaluates to false.	10 == 5	False
		3 == 7	False
		4 == 4	True
!=	<b>Not Equal To</b> - Evaluates as true if the left hand operand is not equal to the right hand operand. Otherwise evaluates to false.	10 != 5	True
		3 != 7	True
		4 != 4	False

The table below shows logical operators.

Operator	Description	Examples	Result
&&	<b>And</b> - Evaluates as true if the left hand operand is true and the right hand operand is true. Otherwise evaluates to false.	True && True	True
		True && False	False
		False && True	False
		False && False	False
	<b>Or</b> - Evaluates as true if the left hand operand is true or the right hand operand is true. Otherwise evaluates to false.	True    True	True
		True    False	True
		False    True	True
		False    False	False
!	<b>Not</b> - Evaluates as true if the operand is false. Otherwise evaluates to false.	!True	False
		!False	True

## L4.1 Ten Guesses

In the solution explorer, navigate to “L4.1-TenGuesses.” and open the .cs file.

In this game, you get ten guesses to guess a random number between 1 and 10 inclusive. We need to add a loop that will repeat ten times. To do this, we’re going to create a while loop. A while loop is a code block (surrounded by { } braces) that repeats while a control value is true.

- First, create a variable to count the number of guesses, which is called a counter in programming.
- Then, while the number of guesses is less than or equal to 10, ask the user to make a guess.
- Whenever the user makes a guess, add one onto the variable that counts guesses.

```
int numberOfGuesses = 0;
int userGuess = 0;

while (numberOfGuesses < 10)
{
    Console.WriteLine("Guess the random number!");
    userGuess = int.Parse(Console.ReadLine());

    numberOfGuesses = numberOfGuesses + 1;
}
```

Add the code above immediately below the line that generates the secret number. Read through the code, then run it. This code takes ten guesses but will never tell the user if their guess is correct. We need to change the code so that the loop runs while the number of guesses is less than ten and the userGuess is not equal to the secretNumber. Let’s change the condition to reflect that.

```
while (numberOfGuesses < 10 && userGuess != secretNumber)
```

Take a moment to read this line of code and understand what it does. The expression in the control statement includes a logical && operator and two relational operators.

The program will end if the user runs out of guesses. What we need to do is give the user feedback, if they won or not. To do this, immediately after the while loop code block, add the following code.

```

if (userGuess == secretNumber)
{
    Console.WriteLine("Congratulations, you win!");
}
else
{
    Console.WriteLine("Sorry, you Lose!");
}

```

Test your code, and once you are satisfied that it works and understand it add the changes to the staging area. Next, commit the changes to the local repository with an appropriate message. Finally, push the changes into GitHub.

## L4.2 N Green Bottles

In the solution explorer, navigate to “L4.2 – NGreenBottles.” and open the .cs file.

Read the code. In this code, you will see an example of a do-while loop, which is a slight variation of the while-loop. When we use a while-loop, the code block is executed if the controlling condition evaluates to true. In a do-while loop, the code block will always be executed at least once, and condition tested at the end of the code block.

```

int count = 10;

do
{
    Console.WriteLine(count + " green bottles standing on a wall.");
    Console.WriteLine(count + " green bottles standing on a wall.");
    Console.WriteLine("but if 1 green bottle should accidently fall");
    Console.WriteLine("there'd be " + count + " green bottles standing on a wall.");
    Console.WriteLine("");
} while (count > 0);

```

Currently, this code runs but results in an infinite loop. To terminate the program in the terminal by pressing Ctrl + C. Then, use what you have learnt to fix the program. If the program works as expected, and terminates the loop appropriately, add the changes to the staging area. Next, commit the changes to the local repository with an appropriate message.

Finally, push the changes into GitHub.

## L4.3 – Celsius-to-Fahrenheit Table

In the solution explorer, navigate to “L4.3 – CelsiusToFahrenheitTable.” and open the .cs file.

Assuming that C is a Celsius temperature, the following formula converts the temperature to a Fahrenheit temperature (F):

$$F = 9/5 * C + 32$$

Create a program that displays a table of the Celsius temperatures from 0 to 10 and their Fahrenheit equivalents. The program should use a for-loop to display the temperatures using "Console.WriteLine" and string Concatenation (+).

In the previous task you probably ended up with some code that looks a little like this:

```
int index = 0;

while (index <= 10)
{
    index++;
}
```

This pattern of creating a control variable (int index = 0) to help iterate over an array, testing some condition to control a loop (index <= 10) and incrementing the control variable (index++;) is so common that there is another type of loop to make this easier, called the for-loop. The for-loop have three elements. The first is the initiator – and it contains code that is executed once, the first time you enter the loop.

```
for (int i = 0; i < 10; i++)
{
}
```

In this case when the loop is initiated a variable with the identifier i is created, and a value of 0 is assigned to that variable. The initiator section can include multiple expressions, but here there is just one.

Next, the condition that controls the loop is defined.

```
for (int i = 0; i < 10; i++)
{
}
```

This must be an expression that evaluates to a Boolean value and controls if the loop code block is executed. Here the loop is controlled by the expression `i < 10`. The last section is the iterator section.

```
for (int i = 0; i < 10; i++)  
{  
  
}
```

This is executed once at the end of each loop. The iterator section can contain multiple expressions, but here it just contains `i++`. `i++` is a unary operator that takes the value in the variable `i` and adds one onto it, so it is equivalent to `i = i + 1`, but `i++` is much more common.

So, to summarise, when the for-loop is executed the initialisation statements are run, creating a variable with the identifier `i` and a value of 0. Next the conditional statement is evaluated. If this evaluates to true the code block following the loop is executed. After the code block is executed the iteration statements are executed. Then the conditional statement is evaluated again. If this evaluates to a value of true the loop is executed again, and if it is evaluated to false the loop ends.

To use the for-loop for this problem you need to establish the initiator's start value (`int i = 0`), condition (`i < 11`) and iterator (`i++`). Then write the for-loop block and for each Celsius value, compute the Fahrenheit value and output the results using `"Console.WriteLine"` and string Concatenation (+), which will all appear within the loop.

Check that your program produces results similar to the one below:

The Celsius to Fahrenheit Conversion Table		
Celsius	Fahrenheit	
0.00	32.00	
1.00	33.80	
2.00	35.60	
3.00	37.40	
4.00	39.20	
5.00	41.00	
6.00	42.80	
7.00	44.60	
8.00	46.40	

Once you have checked your code passes the tests, then add the changes to the staging area. Next, commit the changes to the local repository with an appropriate message. Finally, push the changes into GitHub.

### Formatting the Console.WriteLine results:

Each format item takes the following form and consists of the following components:

```
{index[,alignment][:formatString]}
```

The index is a mandatory parameter specifier, starting from 0 and corresponds to the list of parameters to display in the WriteLine method. The alignment is a signed integer and optional. If the value of alignment is less than the length of the formatted string, then the alignment is ignored, and the length of the formatted string is used as the field width. The formatted data in the field is right-aligned if alignment is a positive value and left-aligned if alignment is a negative value.

### Example:

```
Console.WriteLine("| {0,-10} | {1,5:C2} |", "Name", "Amount");
Console.WriteLine("");
for (int i = 0; i < 5; i++)
{
    string yourName = "John-" + i;
    Console.WriteLine("| {0,-10} | {1,5:C2} |", yourName, i);
}
```

The output will be as below.

Name	Amount
John-0	£0.00
John-1	£1.00
John-2	£2.00
John-3	£3.00
John-4	£4.00

**Note** that the name is left-justified and leaves ten-character spaces because of the sign (-10). The 0 and 1 in the parenthesis correspond to the name and amount parameters, respectively. The C2 means the amount should be formatted as a currency to 2 decimal places, for details on the standard string specifiers, see:

<https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings>

## L4.4 Population Growth

In the solution explorer, navigate to “L4.4–PopulationGrowth.” and open the .cs file.

Create a program that predicts the approximate size of a population of organisms. The program should prompt the user to enter the starting number of organisms, the average daily population increase (as a percentage), and the number of days the organisms will be left to multiply. For example, assume the user enters the following values:

- Starting number of organisms: 2
- Average daily increase: 30%
- Number of days to multiply: 10

The application should display in a tabular format the number of days and current population, see below:

Day	Population
1	2
2	3
3	3
4	4
5	6
6	7
7	10

Note: to avoid integer truncation using cast operator, use either of the following:

- Convert.ToInt32
  - Converts the value of the specified double-precision floating-point number or decimal to an equivalent 32-bit signed integer.
- (int)Math.Round(floatNumber,0)
  - This will achieve the same results as above but uses the Math.Round to round the floating-point number to zero decimal places and cast it to an integer.
  - Math.Round – Rounds a double-precision floating-point value to the nearest integral value and rounds midpoint values to the nearest even number.

### Test data

Start Number	Increase (%)	Days to Multiply													
2	30	3	<table><tr><th>Day</th><th>Population !</th></tr><tr><td>=====</td><td>===== !</td></tr><tr><td>1</td><td>2</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr></table>	Day	Population !	=====	===== !	1	2	2	3	3	3		
Day	Population !														
=====	===== !														
1	2														
2	3														
3	3														
50	25	4	<table><tr><th>Day</th><th>Population !</th></tr><tr><td>=====</td><td>===== !</td></tr><tr><td>1</td><td>50</td></tr><tr><td>2</td><td>62</td></tr><tr><td>3</td><td>78</td></tr><tr><td>4</td><td>98</td></tr></table>	Day	Population !	=====	===== !	1	50	2	62	3	78	4	98
Day	Population !														
=====	===== !														
1	50														
2	62														
3	78														
4	98														
102	17	3	<table><tr><th>Day</th><th>Population !</th></tr><tr><td>=====</td><td>===== !</td></tr><tr><td>1</td><td>102</td></tr><tr><td>2</td><td>119</td></tr><tr><td>3</td><td>140</td></tr></table>	Day	Population !	=====	===== !	1	102	2	119	3	140		
Day	Population !														
=====	===== !														
1	102														
2	119														
3	140														

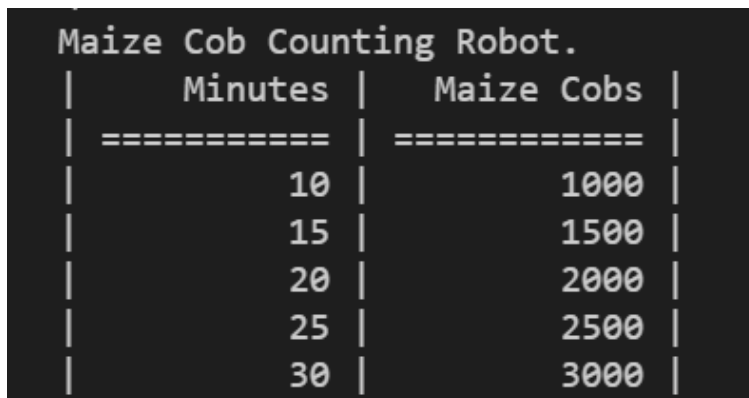


Once you have checked your code passes the tests, then add the changes to the staging area. Next, commit the changes to the local repository with an appropriate message. Finally, push the changes into GitHub.

## L4.5 Maize Cob Robot

In the solution explorer, navigate to “L4.5–MaizeCobRobot” and open the .cs file.

A robot arm at a factory floor collects 100 maize cobs per minute. The robot has been install with a display device, which updates every five minutes start from the tenth minute. Create a program that emulates the robots display. The program should use for-loops to display in the console the number of maize cobs collected after 10, 15, 20, 25, and 30 minutes. The output of the program should look like the one below.



Minutes	Maize Cobs
10	1000
15	1500
20	2000
25	2500
30	3000

Note that in the for-loop, you need to establish the initiator’s start value, condition and iterator.

Once you complete the codes, check that your results correspond to the above and if it does, add the changes to the staging area. Next, commit the changes to the local repository with an appropriate message. Finally, push the changes into GitHub.

## Assessment Block

*L4.6-Bacterial Frequency Table and L4.7-The Palindrome Dice Game are part of your portfolio submission (subject to approval) and will be assessed towards the end of the trimester. For further details on the submission, see the assignment brief (coming up shortly).*

### L4.6-Bacterial Frequency Table

A scientist measures the lengths of bacteria in nanometres used in an experiment. The total number of bacteria measured for the experiment is unknown at the beginning of the data collection. However, their lengths are in the ranges of 1 – 800 ( $1 \leq \text{length} < 800$ ) nanometres. The scientist wants to collect data based on the following ranges:

- < 200 nanometres
- 200 – 399 nanometres
- 400 – 599 nanometres
- 600 – 799 nanometres

The scientist wants to produce a frequency distribution table from the data collected, which eventually will be used to construct a histogram.

Write a program that allows the scientist to enter the lengths of the bacteria until the number entered is greater than the upper limit of the specified range ( $1 \leq \text{length} < 800$ ). When this happens, the program should out the frequency distribution below:

Range	Count	Sum Lengths	Percentage
=====	=====	=====	=====
<200	10	783	30.30%
200 - 399	7	2056	21.21%
400 - 599	8	3986	24.24%
600 - 799	8	5552	24.24%
The mean length of bacteria: 375.06			
The Maximum length of bacteria: 799.00			
The Minimum length of bacteria: 12.00			

Do not use arrays to store the numbers, but process the numbers on the fly. Below are some additional notes:

- “Count” is the number of bacteria lengths entered for each range.
- “Sum Lengths” is the total length of all bacteria in each range.
- “Percentage” is the range count divided by total counts across all ranges, expressed as a percentage.

- The mean length of bacteria is the total length divided by the number of bacteria across all ranges.
- The maximum length of bacteria is the longest lengths recorded.
- The minimum length of bacteria is the shortest lengths recorded.

Generate your own test data to test your program, and add this data together with the results as comments in you program.

Once you have checked your code passes the tests, then add the changes to the staging area. Next, commit the changes to the local repository with an appropriate message. Finally, push the changes into GitHub.

## L4.7-The Palindrome Dice Game

The Palindrome Dice game is a number game which is played by a single players.

The Game is played with three dices, which are tossed simultaneously and used to form a three digit number. For example, if the three dice return 4, 7, 4 respectively, the three digit number form is 474.

The formed three digit number is then check to see if it is a palindrome number. If it is then the player earns a score of 10. The player is then prompted to double that score. If they decide to double the score, then the three dice is cast again, and a three digit number like the one mentioned above is formed. If the three digit number is a palindrome the players score of 10 is double, otherwise the player loses that score. The game console is cleared and a play statistics like the one below is displayed.

```
=====
*****Game Statistics*****
=====
Number of Palindromes: 20
Number of Throws: 100
Game Score: 170
=====
```

The game ends when the number of times you've tossed the three dice reaches a 100. You will win the game if your score is greater than the number of throws, otherwise you lose. After the game is complete the play statistics is displayed and a message indicating a win or lose is played, see example below:

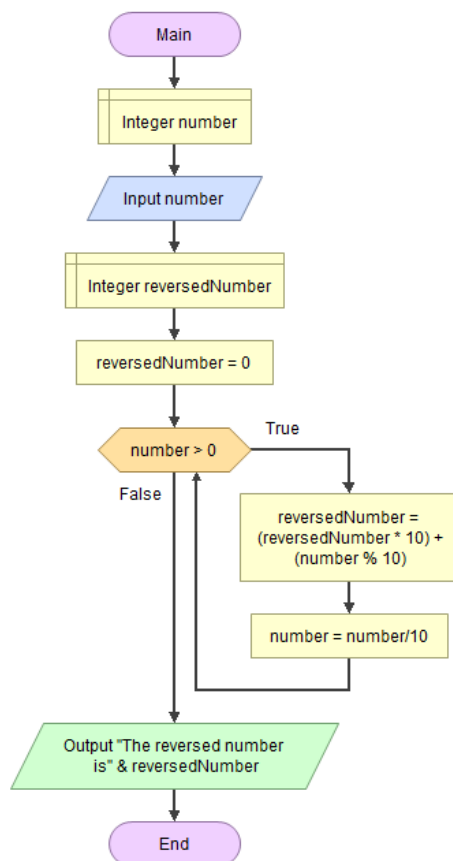
```

=====
*****Game Statistics*****
=====
Number of Palindromes: 20
Number of Throws: 100
Game Score: 170
=====
You WON the game!

```

## Additional Information

A palindromic number (also known as a numeral palindrome or a numeric palindrome) is a number (such as 121) that remains the same when its digits are reversed. In other words, it has reflectional symmetry across a vertical axis. See the flowchart below for reversing a number.



Once you have checked your code passes the tests, then add the changes to the staging area. Next, commit the changes to the local repository with an appropriate message. Finally, push the changes into GitHub.

## Summary

In this lab we've looked at repeating code using while, do while loops and for-loops. We have also seen how to format the string output in the console (`Console.WriteLine`) using some standard numeric string formats.