PROGRAMMING WITH

An introduction

KES Summer 2023

v1.1

Contents

Introduction 3

What is C#? 3

Visual Studio........ 4

Item properties 4

Project files 4

Syntax errors 4

Code editor 4

Basic Data types 10

Variables 11

Getting user input 13

Strings 17

Building a User Interface (UI) 28

A bit of maths 44

Selection 60

Why use an array 74

Iteration 77

Logical Operators 86

Advanced strings and loops 94

Subroutines - Creating a program structure 98

Exception handling 106

Robust subroutine code 113

Putting it all together – Complete code example 118

Resources 126

INTRODUCTION

This book will focus on developing programming skills and will go through the core concepts of syntax and the main building blocks of programming, which are sequence, selection and iteration. The algorithms for each program will be described using pseudo code, so that you can try the programs with other languages in the future if you wish.

WHAT IS C#?

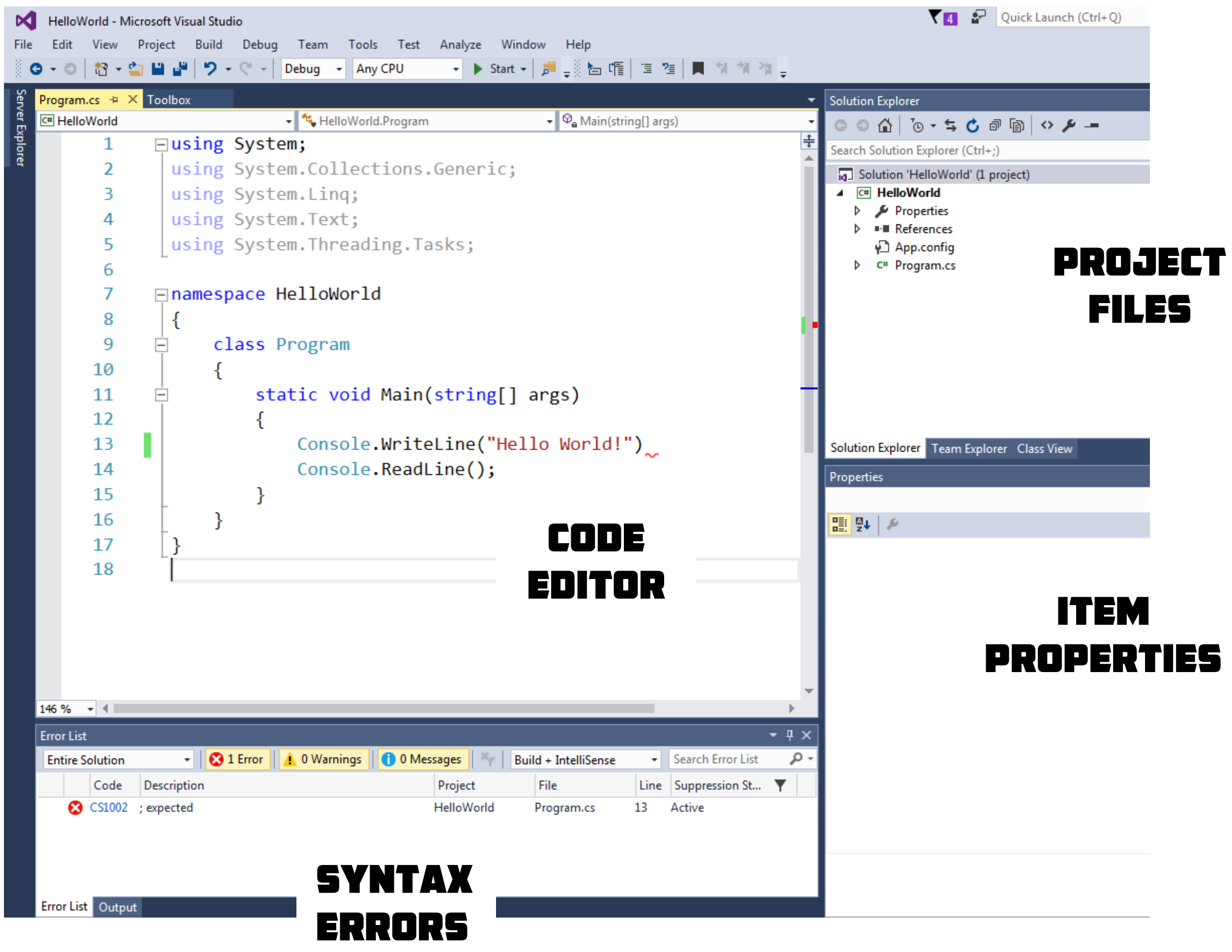
C# is a high-level programming language developed by Microsoft and is part of the .NET framework. The idea behind a software framework is to group all the tools together to help a programmer become more productive. It contains a large code library to help you with common tasks; this allows you to rapidly develop software applications. C# is considered to be similar to Java and other modern programming languages.

With C# you can create:

- Windows applications
- Dynamic web pages
- Mobile applications
- Games
- Data manipulation applications
- Simulations e.g. science

VISUAL STUDIO

Visual Studio is a software application for writing software. It provides a range of tools to help developers create and test their software.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!")
            Console.ReadLine();
        }
    }
}
```

Error List

Entire Solution | 1 Error | 0 Warnings | 0 Messages | Build + IntelliSense | Search Error List

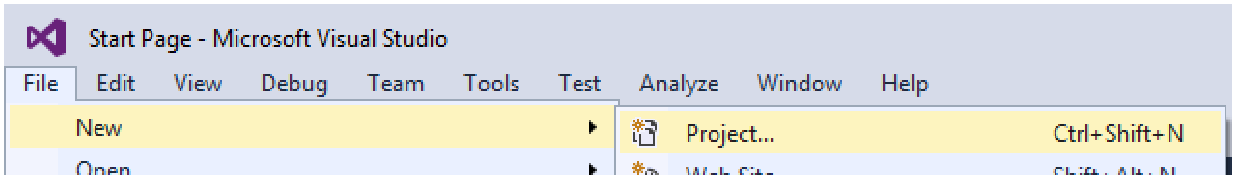
	Code	Description	Project	File	Line	Suppression St...
❌	CS1002	; expected	HelloWorld	Program.cs	13	Active

GETTING STARTED

WRITING YOUR FIRST PROGRAM

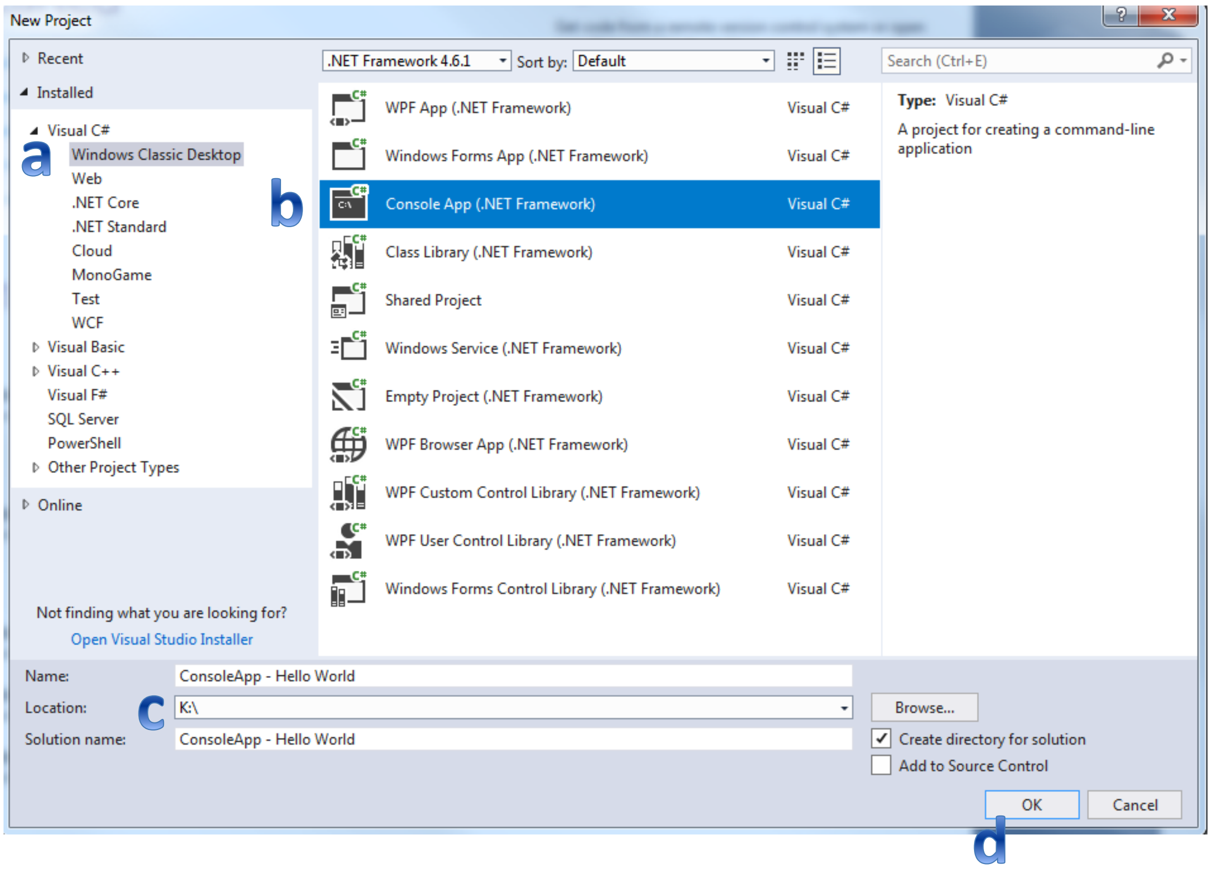
The best way to learn programming is to program, so let's create a simple text based program using Visual Studio with C#.

1. CREATE A NEW PROJECT


Start Page - Microsoft Visual Studio
File Edit View Debug Team Tools Test Analyze Window Help
New
Project... Ctrl+Shift+N

2. C# CONSOLE APPLICATION

3. A C# console application is the simplest type of program and will produce a text based program.
   a) Choose Visual C#, Windows Classic Desktop
   b) Console Application (.NET Framework)
   c) Change the name to 'ConsoleApp - Hello World'
   d) Select OK to create


New Project

.NET Framework 4.6.1 — Sort by: Default — Search (Ctrl+E)

Recent
Installed
Visual C#
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
MonoGame
Test
WCF
Visual Basic
Visual C++
Visual F#
SQL Server
PowerShell
Other Project Types
Online

Not finding what you are looking for?
Open Visual Studio Installer

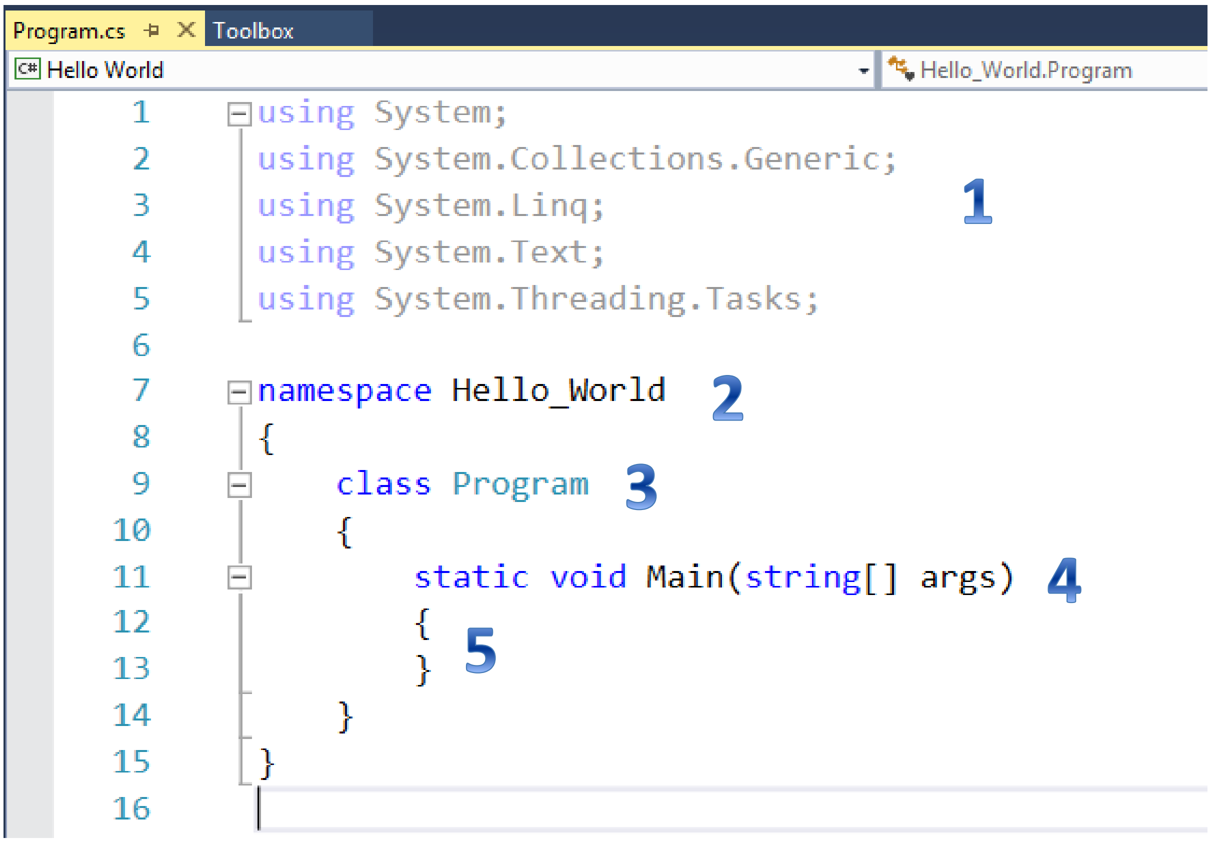
WPF App (.NET Framework)	Visual C#
Windows Forms App (.NET Framework)	Visual C#
Console App (.NET Framework)	Visual C#
Class Library (.NET Framework)	Visual C#
Shared Project	Visual C#
Windows Service (.NET Framework)	Visual C#
Empty Project (.NET Framework)	Visual C#
WPF Browser App (.NET Framework)	Visual C#
WPF Custom Control Library (.NET Framework)	Visual C#
WPF User Control Library (.NET Framework)	Visual C#
Windows Forms Control Library (.NET Framework)	Visual C#

Type: Visual C#
A project for creating a command-line application

Name: ConsoleApp - Hello World
Location: K:\
Solution name: ConsoleApp - Hello World
Browse...
[✓] Create directory for solution
[ ] Add to Source Control
OK Cancel

STRUCTURE OF A C# PROGRAM

Visual Studio will create all the necessary code for you to begin. You don't need to worry about most of this for now, just where to type your code, which is where 5 is.



```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace Hello_World
 8  {
 9      class Program
10      {
11          static void Main(string[] args)
12          {
13          }
14      }
15  }
16
```

1. The **using** statements are called **directives**. They tell the program which pre-made code library namespaces are needed. For now, allow Visual Studio to manage this for you.

   For example, the **using System** directive allows the user to interact with the program using the pre-built library code provided e.g. WriteLine()

2. A **namespace** is used to group code together in a name given by the programmer

3. The **class** section is used to group code statements together, the programmer can name and categorise each section of the code. This will be explained later.

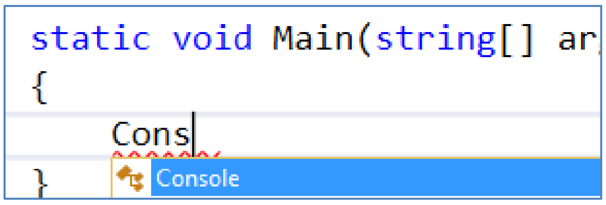
4. The **Main()** section is called a **method** and is where the program starts.

5. The {} curly brackets is where you will type you code. The code will run in sequence from line 12.

INTELLISENSE

Typing code into Visual Studio is easy once you get started.

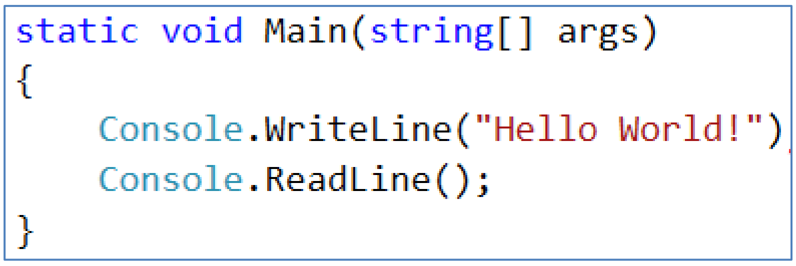
Just start typing and choose what you want from the list, this is the ***intellisense*** feature.



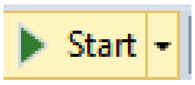
```
static void Main(string[] ar
{
    Cons
}
```

ACTIVITY 1 : CODE TIME

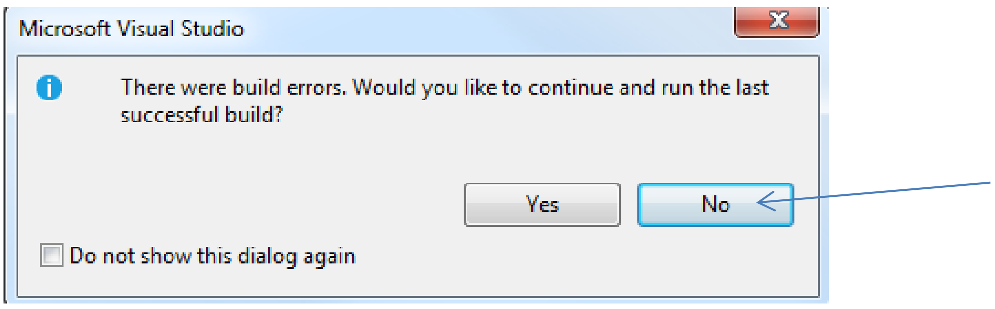
1. Type the following code:



```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!")
    Console.ReadLine();
}
```

2. Press  to run the code

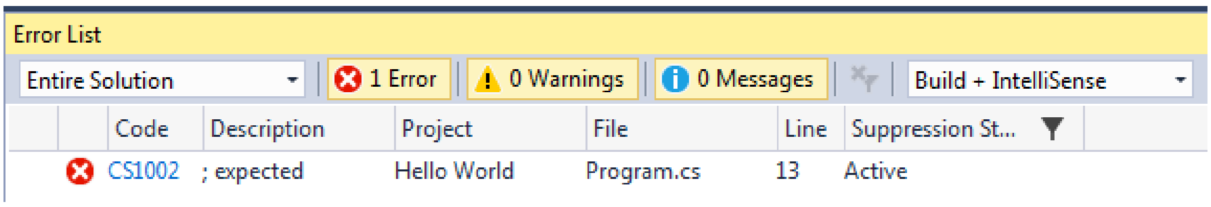
3. There is an error, so you will get a message. Press **NO**



Microsoft Visual Studio

There were build errors. Would you like to continue and run the last successful build?

4. An error list will appear at the bottom, you are missing a **;**

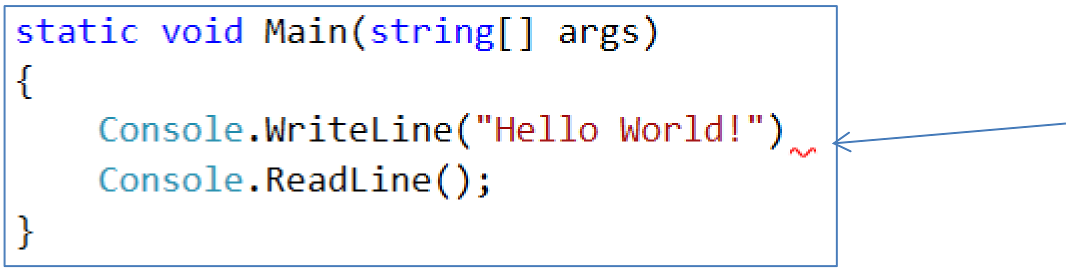


Error List

	Code	Description	Project	File	Line	Suppression St...
✖	CS1002	; expected	Hello World	Program.cs	13	Active

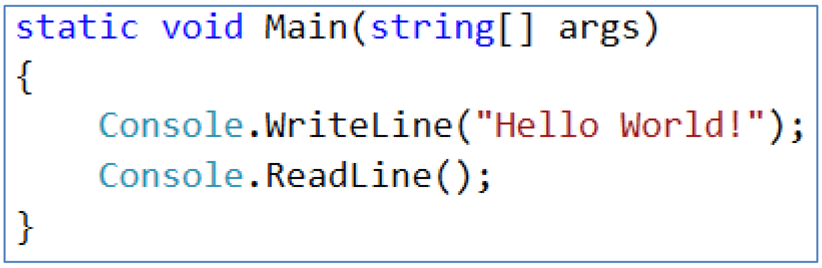
5. Double click on the error line and the cursor will jump to the correct place.

   You may have noticed a red ~ this is where the error was

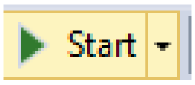


```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!")
    Console.ReadLine();
}
```

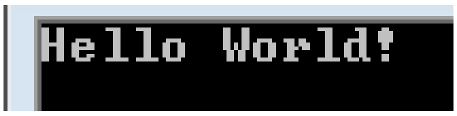
6. Type a ; to complete the code and the line below



```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
    Console.ReadLine();
}
```

7. Press  to run the code

8. It should run and show you:



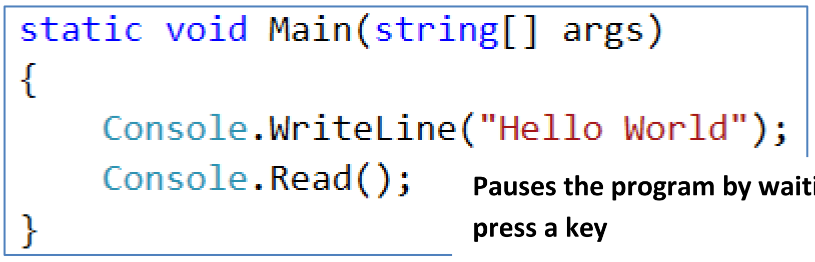
```
Hello World!
```

9. Press enter to close the program.

ACTIVITY 1 : CODE SENSE

Great! It worked but what did it do?

Here is a brief explanation:



```
static void Main(string[] args)
{
    Console.WriteLine("Hello World");
    Console.Read();
}
```

Displays Hello World on the screen

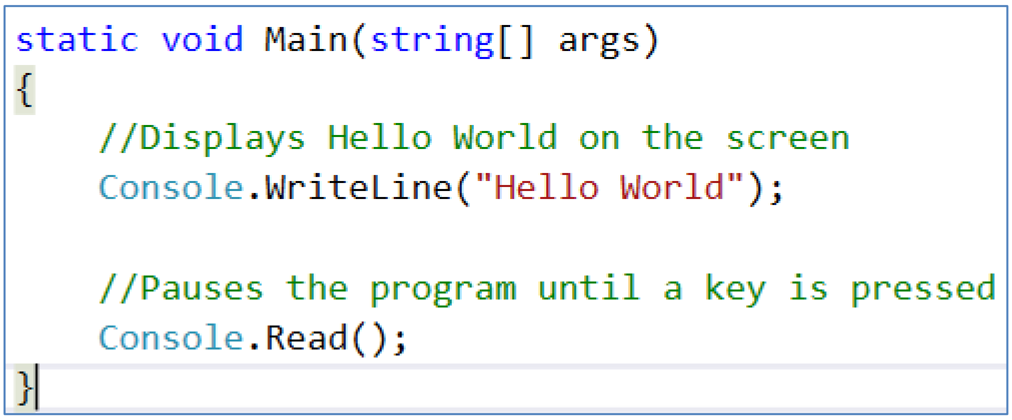
Pauses the program by waiting for the user to press a key

ACTIVITY 1 : CODE COMMENTS

It is good practise to add comments to you program so other know what it is doing or just for your own benefit.

// is how you tell C# that it is a comment and not to run it as code.

Add comments to you program



```
static void Main(string[] args)
{
    //Displays Hello World on the screen
    Console.WriteLine("Hello World");

    //Pauses the program until a key is pressed
    Console.Read();
}
```

Well done, you have created your first program.

BASIC DATA TYPES

Now you have written your first program lets explore some key concepts. When programming, you will be working with different types of data. All programming languages can use different data types and the common ones are:

INTEGER

An integer is a whole number (no fractional parts).	*For example:* 1, 2, 125, 4280, -134	*In C# this is defined as:* **int**	*Memory usage:* 4 bytes

REAL OR FLOAT

Real or floating point numbers have a decimal place and can represent a fraction.	*For example:* 12.8, 128.967, -1.2342	*In C# this is defined as:* **float**	*Memory usage:* 8 bytes

CHAR OR CHARACTER

Char stands for character; a character is a letter, punctuation mark, symbol or digit.	*For example:* ‘A’, ‘b’, ‘?’, ‘@’, ‘4’	*In C# this is defined as:* **char**	*Memory usage:* 1 byte per character

STRING

A string is a piece of text, multiple characters strung together. Actually a string can be zero (empty), one or more characters.	*For example:* “” (empty) “Hello world!” “H”	*In C# this is defined as:* **string**	*Memory usage:* 1 byte per character

BOOLEAN

A Boolean can only store true or false	*For example:* true false	*In C# this is defined as:* **bool**	*Memory usage:* 1 byte

VARIABLES

A variable is a named location in memory. Variables are used to store values that are being used by the program in a computers memory. When a program runs it can store, change and access the data.

IDENTIFIER

A variable can be identified by its name or identifier. When asked for the identifier for a variable you are being asked for its name.

Programming languages have rules for naming variables.

NAMING VARIABLES

Variable names in C# must follow these rules:

1. The first character cannot be a number
2. There cannot be spaces between words
3. The name can only contain letters, numbers or underscores (_)
4. The name cannot use certain reserved words e.g. Console, namespace, class, if, for, while and many more
5. The name is case SEnSiTIVE e.g. userage is not the same as USERAGE or userAge

NAMING CONVENTIONS

Although not a rule, there are two main methods that programmers use to name variables when using multiple words (see rule 2):

CAMEL CASE

When using camel case (because it has humps!) the first letter is lowercase, and then each following word is capitalised:

- userName
- numberOfAttendees

UNDERSCORE (_)

When using the underscore method again all words are lowercase but are joined by an underscore:

- user_name
- number_of_attendees

SETTING UP C# VARIABLES

It is important to plan your programs before writing the code, especially the variables. You need to consider how they will be used and what data types will be needed.

When setting up or declaring a variable in C#, you need to:

1. identify the data type
   - int, float, char, string or bool

2. provide an identifier
   - Use the naming rules and conventions

3. assign an initial value
   - This can be zero or an empty string "" ready to store a new value as the program runs

ASSIGNING A VALUE

When you give a value to a variable it is called assigning a value. Some languages require a value to be given to a variable when it is initially setup, this is called initialisation.

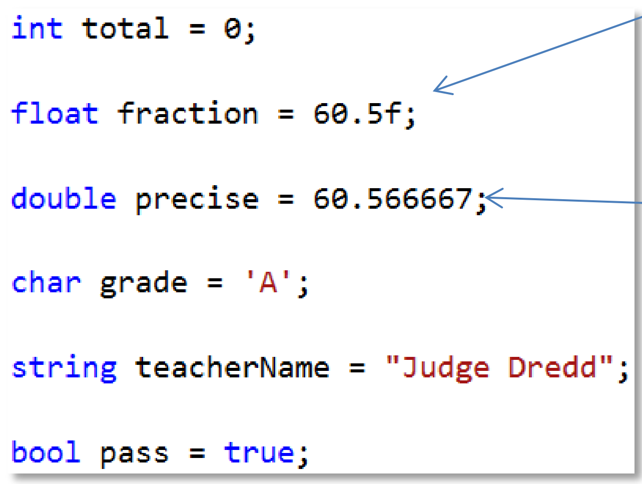
In pseudocode the ← symbol is used to indicate when to assign a value to a variable, for example:

total ← 0

Note: In pseudocode the = symbol means make a comparison.

In C# values are assigned using the = symbol

Examples



```
int total = 0;

float fraction = 60.5f;

double precise = 60.566667;

char grade = 'A';

string teacherName = "Judge Dredd";

bool pass = true;
```

Note: A **float** data type requires an **f** after the value.

This is to tell C# to store the number as a floating point decimal format.

C# can store decimals in different formats, like a double.

GETTING USER INPUT

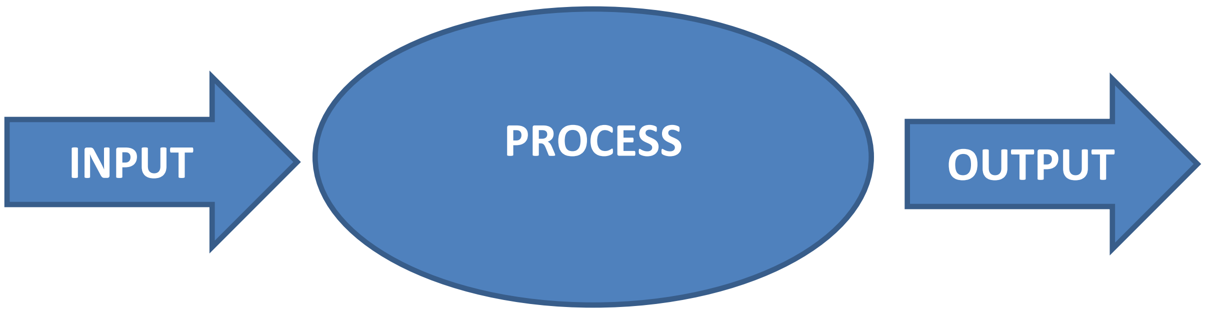
Now that you have written your first program and (hopefully) understand what is going on, let’s get the user involved.

INPUT, PROCESS AND OUTPUT

When programming I like think about the program in simple terms, namely;

Input, Process and Output.

This will help you plan your programs.



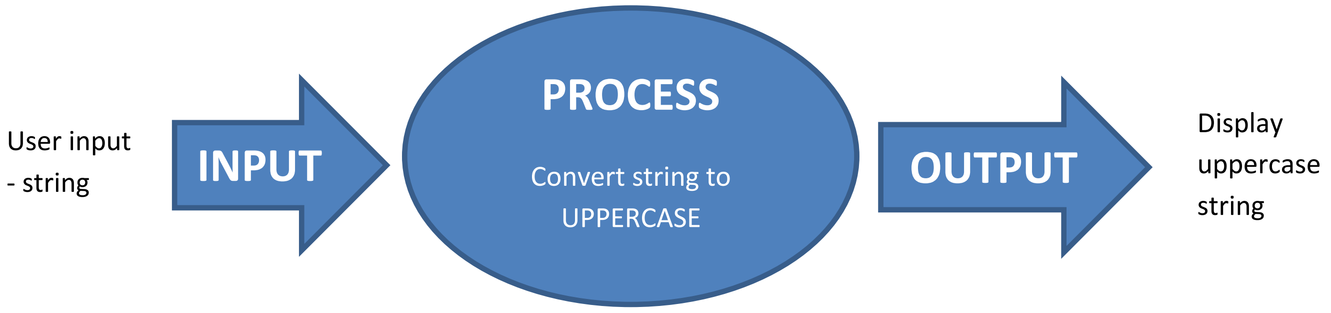
This idea can also be used differently later when you are using more complex programming techniques.

ACTIVITY 2 : USER INPUT TO UPPERCASE

The following program will take in a user input as string, make it upper case and output it to screen.

I-P-O DIAGRAM

Here is a simple Input-Process-Output diagram to describe the program, it gives a simple overview, rather than how to write the program.



PSEUDOCODE

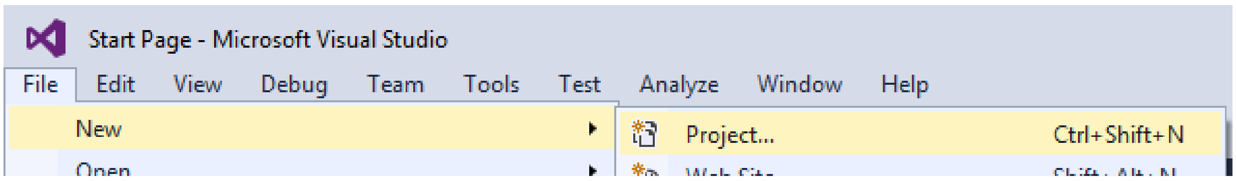
Here is a pseudocode version of the same program, as you can see it is a guide on how the program could be written.

This design is then applied to the language of your choice, in this case C#.

```
OUTPUT "Hello, please enter a word: "
wordUserInput ← USERINPUT
#Change the word to upper using a built-in function / method
wordUpper ← UPPER(word)
OUTPUT "The word entered is now uppercase: "
OUTPUT wordUpper
```

ACTIVITY 2 : CODE TIME

1. CREATE A NEW PROJECT

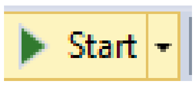

Start Page - Microsoft Visual Studio
File Edit View Debug Team Tools Test Analyze Window Help
New
Open
Project... Ctrl+Shift+N

2. C# CONSOLE APPLICATION

3. A C# console application is the simplest type of program and will produce a text based program.
   a) Choose Visual C#, Windows
   b) Console Application
   c) Change the name to 'ConsoleApp - User input to uppercase'
   d) Select OK to create

4. Type the following code:

```
static void Main(string[] args)
{
    string wordUserInput = "";
    string wordUpper = "";

    Console.Write("Please enter a word: ");
    wordUserInput = Console.ReadLine();
    wordUpper = wordUserInput.ToUpper();
    Console.WriteLine("The word entered is now uppercase: ");
    Console.Write(wordUpper);
    Console.Read();
}
```

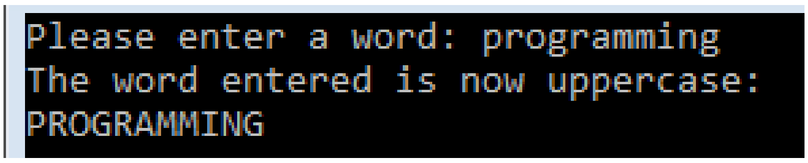


e) Press to run the code

5. If there are errors, check your syntax, fix and try again.

   Common errors:
   a) Not initialising a variable with a value e.g. wordUserInput **= ""**
   b) Forgetting to end each statement with a **;**
   c) Forgetting to add a second **"** or **)**
   d) Not adding **()** to the end of a function/method e.g. ToUpper**()**
   e) Deleting the closing **}** at the end of the program

6. It should run and shown you first simple program.



```
Please enter a word: programming
The word entered is now uppercase:
PROGRAMMING
```

7. Press enter to close the program.

ACTIVITY 2 : CODE SENSE

Super! It worked but what did it do?

Here is a brief explanation:

```
static void Main(string[] args)
{
    string wordUserInput = "";
    string wordUpper = "";
```

Set up two string variables and initialise with an empty string value ""

```
    Console.Write("Please enter a word: ");
```

Display "Please enter a word" on screen and keep cursor on the <u>same line</u>

```
    wordUserInput = Console.ReadLine();
```

Read a line of characters input by the user and store the value in the worduserInput variable

```
    wordUpper = wordUserInput.ToUpper();
```

Convert the value in the worduserInput variable to UPPERCASE and store in the wordUpper variable

```
    Console.WriteLine("The word entered is now uppercase: ");
```

Display "the word entered is now uppercase: " and move the cursor <u>down</u> a line

```
    Console.Write(wordUpper);
```

Display the value of the wordUpper variable on screen

```
    Console.Read();
}
```

Pauses the program by waiting for the user to press a key, before closing the program

Note: Console.Readline() always reads the user input as a string, even if they enter an integer! More on this later.

ACTIVITY 2 : CODE COMMENTS

Use // to add comments to your code to help you in the future.

Don't forget to save!

STRINGS

A **string** is a piece of text, multiple characters strung together, they can be a single word, multiple words with spaces or empty.

Strings are a vital part of many programs and come with many properties and built-in functions / methods that can use used.

ASSIGNING A STRING TO A VARIABLE

You have already done this but a quick recap might be useful.

To assign a string to a variable first the variable must be setup (declared) and then assigned a value using the = symbol.

INITIALISE A VARIABLE

You can initialise a variable by assigning an empty string:

```
string name = "";
```

You can initialise a variable by assigning a string value:

```
string name = "A Programmer";
```

UPDATE A VARIABLE

You can also update a string with another value by using the = symbol again:

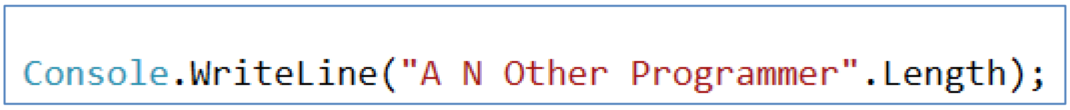
```
string name = "";

name = "A N Other Programmer";
```

FINDING THE LENGTH OF A STRING

Strings have **properties** available and one of those is its ***Length***.

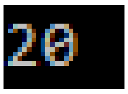
The *Length* function / method can be used in the following ways, but is most commonly used with a variable.

Directly with a string



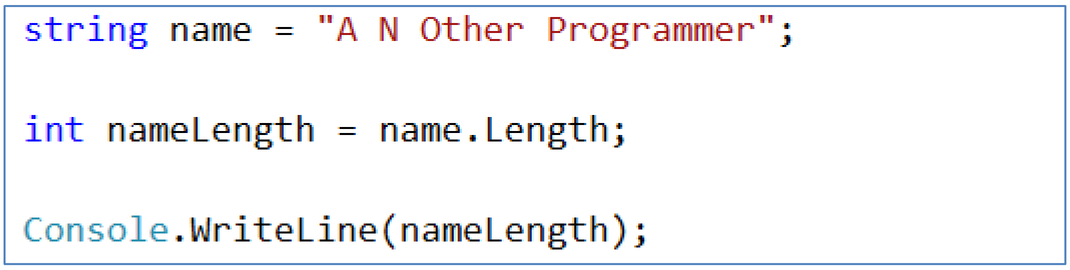
```
Console.WriteLine("A N Other Programmer".Length);
```

Output:



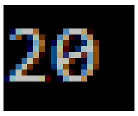
```
20
```

Using a variable containing a string value



```
string name = "A N Other Programmer";

int nameLength = name.Length;

Console.WriteLine(nameLength);
```

Output:



```
20
```

Note: ***.Length*** is a property and not a method so doesn't need () after the key word.

STRING INDEX

Every character within a string has a position number, which is known as the index. The number starts at zero.

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
String	**P**	**r**	**o**	**g**	**r**	**a**	**m**	**m**	**e**	**r**

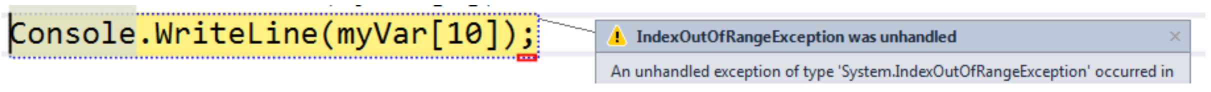
The index of a string can be used in various ways and works well with variables.

To identify a letter from a string VariableName[index Number] is used.

Example

```
string myVar = "Programmer";

Console.WriteLine(myVar[1]); //produces the letter 'r'

Console.WriteLine(myVar[2]); //produces the letter 'o'

Console.WriteLine(myVar[10]); //will produce an error!
```

Even though "Programmer" has 10 letters, the index starts at 0 and ends at 9. Be careful!



```
Console.WriteLine(myVar[10]);
```

IndexOutOfRangeException was unhandled

An unhandled exception of type 'System.IndexOutOfRangeException' occurred in

Note: The error ***IndexOutofRangeException*** means that the code asked for an index value of 10, which is beyond the size or range of the index positions 1-9. Simply put there isn't an index position 10.

WAYS OF USING THE STRING INDEX

Here are two alternative ways of using the index position in a program.

Using variables to store the result

Using a variable to store a character is useful if you need to use the result again later in the program.

```
string name = "C Sharpe";
char initial = name[0];

Console.WriteLine(initial);
```

Did you notice the data type?

In C# a variable with single character is a char

Straight to output

Sometimes you might just want to output the result once, here you could consider producing the result directly in Console.WriteLine().

```
string name = "Programmer";

Console.WriteLine("The initial is " + name[0]);

Console.Read();
```

JOINING STRINGS TOGETHER

STRING CONCATENATION

Joining two strings together is called **concatenation**.

The concatenate symbol (**+**) can be used to join two strings or two variables that contain strings.

Example

```
Console.WriteLine("Once there was a " + "giant");
//is the same as, so not that useful
Console.WriteLine("Once there was a giant");

//More useful is joining a string with a variable
string character = "unicorn";
Console.WriteLine("Once there was a " + character);
```

Joining strings with other data types

In this example the noHorns variable is an integer. It works no problem.

```
string character = "unicorn";
string mood = "sad";
int noHorns = 0;

Console.WriteLine("Once there was a " + mood + character + ".");
Console.WriteLine("Who had " + noHorns + " horns");
```

PLACE HOLDERS

Placeholders and strings

Strings can also be constructed by using place holders, slightly more complex to use but very powerful.

Placeholders use an index number and work in a very similar way to a string index with {indexNumber} being used to identify where to place each variable in the string.

Example

```
string character = "giant";
string mood = "grumpy";

Console.WriteLine("The {0} was {1}.", character, mood);
```

{0} ***{1}***

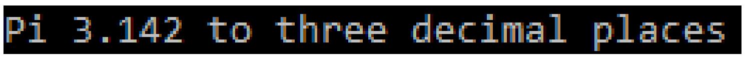
index number

Placeholders and numbers

When using a number in a string they can be formatted using a place holder. Here is an example changing the number of decimal places.

index number 0 : Floating point 3 decimal places

```
Console.WriteLine("Pi {0:F3} to three decimal places", 3.14159265359);
```



```
Pi 3.142 to three decimal places
```

ESCAPE CHARACTERS

Escape characters are used to format or output special characters that don’t work as expected unless given extra information.

To allow certain characters to display a backslash (\) needs to be added.

SINGLE OR DOUBLE QUOTES

```
Console.WriteLine("\"Double quotes\"");
Console.WriteLine("\'Single quotes\'");
```

```
"Double quotes"
'Single quotes'
```

BACKSLASH

```
Console.WriteLine("\\ backslash");
```

```
\ backslash
```

NEWLINE

```
Console.WriteLine("new \nline");
```

```
new
line
```

TAB

```
Console.WriteLine("\t tab with a space at the start");
Console.WriteLine("\ttab no space");
```

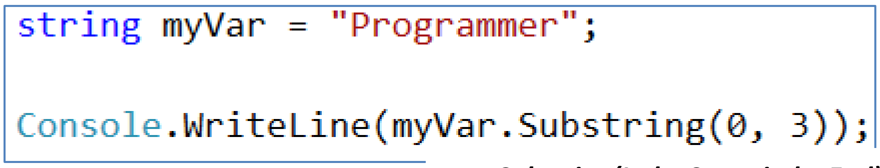
```
         tab with a space at the start
        tab no space
```

FINDING PART OF A STRING

Finding part of a string is also known as a **substring**.

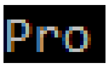
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
String index	**P**	**r**	**o**	**g**	**r**	**a**	**m**	**m**	**e**	**r**

As you know, every character within a string has a position number, which is known as the index. Instead of just getting a single character, multiple characters can be extracted by giving an index start and index end position.



```
string myVar = "Programmer";

Console.WriteLine(myVar.Substring(0, 3));
```

. Substring(IndexStart, indexEnd)



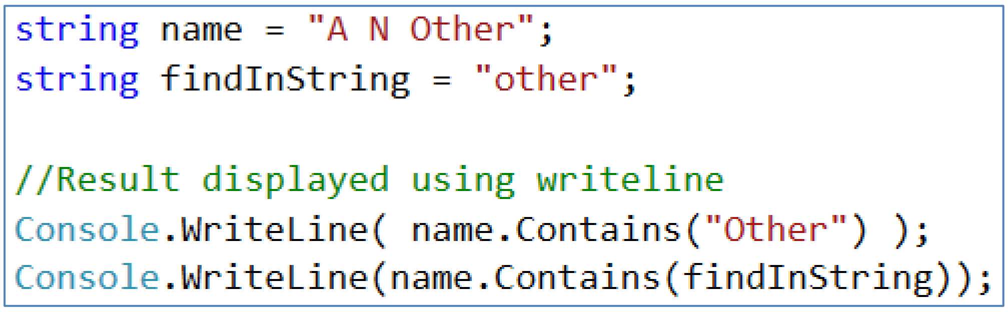
```
Pro
```

A FINDING A STRING INSIDE A STRING

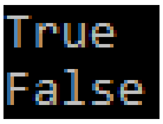
Sometimes you want to check inside a string to see if another string is part of the text.

Comparisons are **case sensitive** and can be made between two variables or a variable and string value.

String.Contains("stringToFind") will return true or false, depending whether the string is found.



```
string name = "A N Other";
string findInString = "other";

//Result displayed using writeline
Console.WriteLine( name.Contains("Other") );
Console.WriteLine(name.Contains(findInString));
```



```
True
False
```

Note: False occurs because "other" is all lowercase whereas "**O**ther" has a capital "**O**".

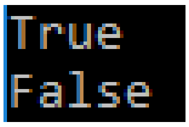
COMPARING STRINGS

Sometimes you may need to compare two strings to see if they are match or not. C# provides a function / method to help you do this.

Comparisons are case sensitive and can be made between two variables or a variable and string value.

If a match is found it will output **true**, if not **false**.

```
string password = "scadoosh";
string check = "SCADOOSH";

//Result displayed using writeline
Console.WriteLine( password.Equals("scadoosh") );
Console.WriteLine( password.Equals(check) );
```



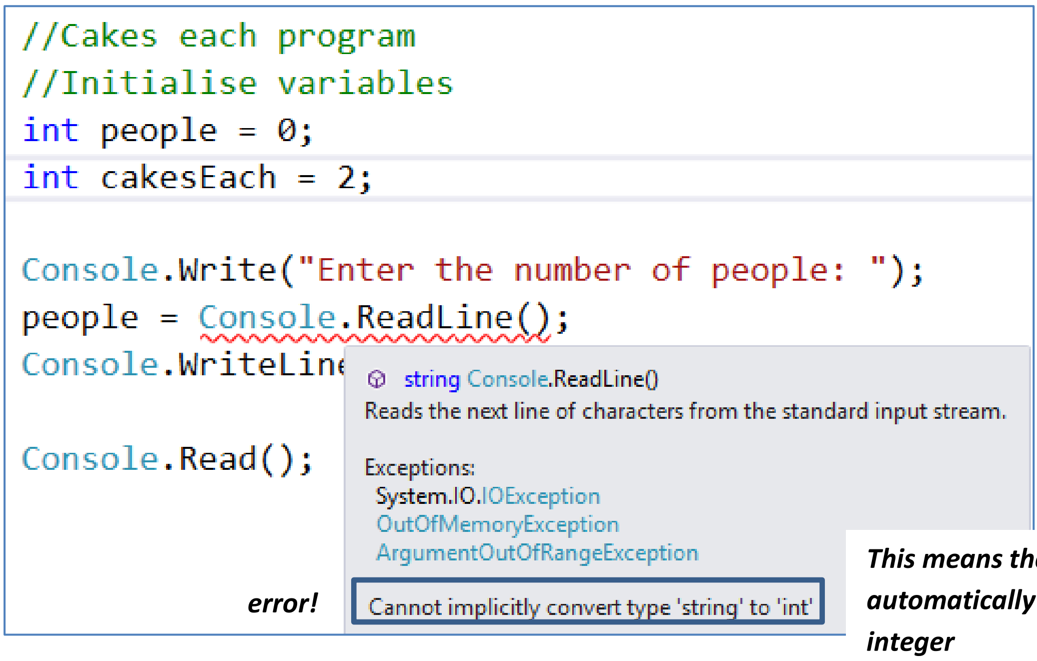
```
True
False
```

CHANGING DATA TYPE AKA TYPE CASTING

If you remember from before when getting user input it is always a string. This is fine if you want to output this number to a string. Often, this is not the case and you will want to use the input as an integer to perform a calculation.

Example that doesn't work

The following example doesn't work as even though the people variable is a number.



```
//Cakes each program
//Initialise variables
int people = 0;
int cakesEach = 2;

Console.Write("Enter the number of people: ");
people = Console.ReadLine();
Console.WriteLine

Console.Read();
```

string Console.ReadLine()
Reads the next line of characters from the standard input stream.

Exceptions:
System.IO.IOException
OutOfMemoryException
ArgumentOutOfRangeException

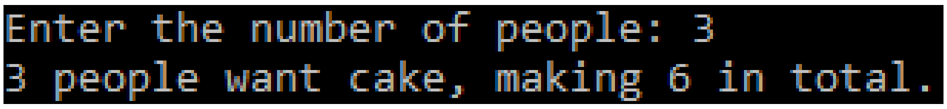
error!

Cannot implicitly convert type 'string' to 'int'

This means that C# cannot automatically change a string to an integer

Example that does work

```
//Cakes each program
//Initialise variables
int people = 0;
int cakesEach = 2;

Console.Write("Enter the number of people: ");
people = int.Parse(Console.ReadLine());
Console.WriteLine("{0} people want cake, making {1} in total.", people, cakesEach*people);
```



```
Enter the number of people: 3
3 people want cake, making 6 in total.
```

Did you spot the difference? the function / method **int.Parse()** analyses at the user input and checks to see if it can be change into an integer.

```
people = int.Parse(Console.ReadLine());
```

Parse can also be used with **float**, **double** and **decimal**.

PRACTICE ACTIVITY: WRITE A SHORT STORY

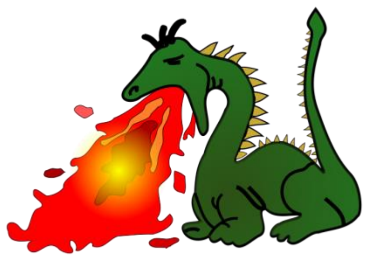
Write a short story with a basic structure but ask the user to fill in some gaps. Don't make it too complicated but this is your chance to use your imagination and be creative.

The story should ask a user to:

- Identify a character type e.g. giant, unicorn, werewolf, sponge
- Give the character a name, Title, forename and surname
- Identify the age of the character as an integer
- Add any other parts of the story that you would like to ask the user

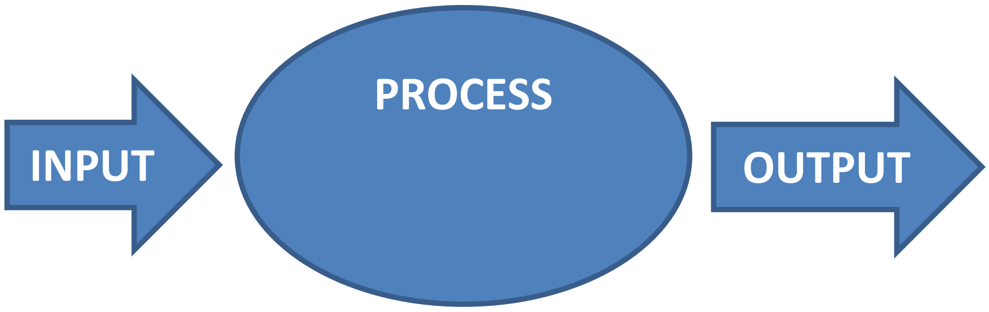
The story should:

- Store the user input in variables of the correct data type
- Join strings together using concatenation, placeholders or both
- Use an escape character
- Use part of a string (substring)
- Use type casting to convert a user input to a number



DON'T FORGET TO PLAN YOU PROGRAM

I-P-O



PSEUDOCODE

Here is a possible starting point.

```
OUTPUT “Please enter a type of fictional animal: “
fictionalAnimal ← USERINPUT
etc.…
#Change to upper using a built-in function / method
fictionalAnimalUpper ← UPPER(fictionalAnimal)
etc.…
OUTPUT “Once there was a “ + fictionalAnimal
OUTPUT “That used to like shouting. ”
#Display a double quote using an escape character
OUTPUT “\”I am a ” + fictionalAnimalUpper + “\””
etc.…
```

BUILDING A USER INTERFACE (UI)

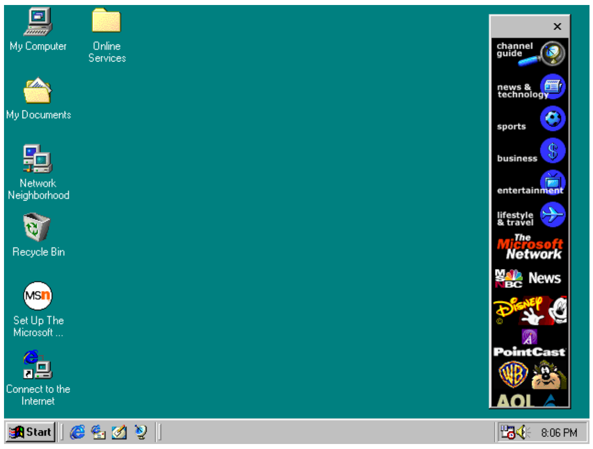
In this section you will build you first user interface. The type of software application that most users will be familiar with have a **user interface** designed to make using the application easier. A UI is how the user and a computer system interact

Designing a user interface is an entire topic on it's own, you will just cover the basics.

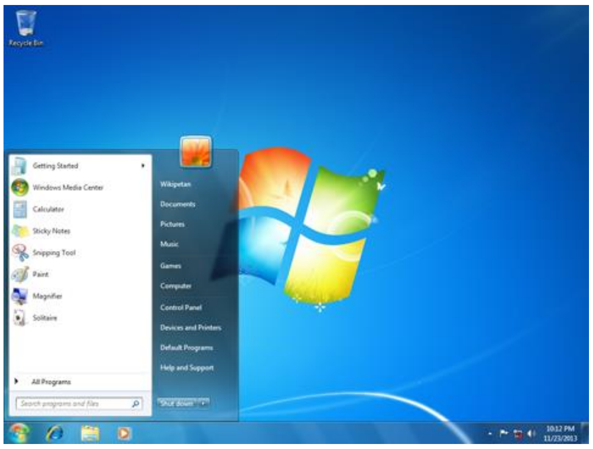
EXAMPLES

Operating System Graphical User Interface (GUI)

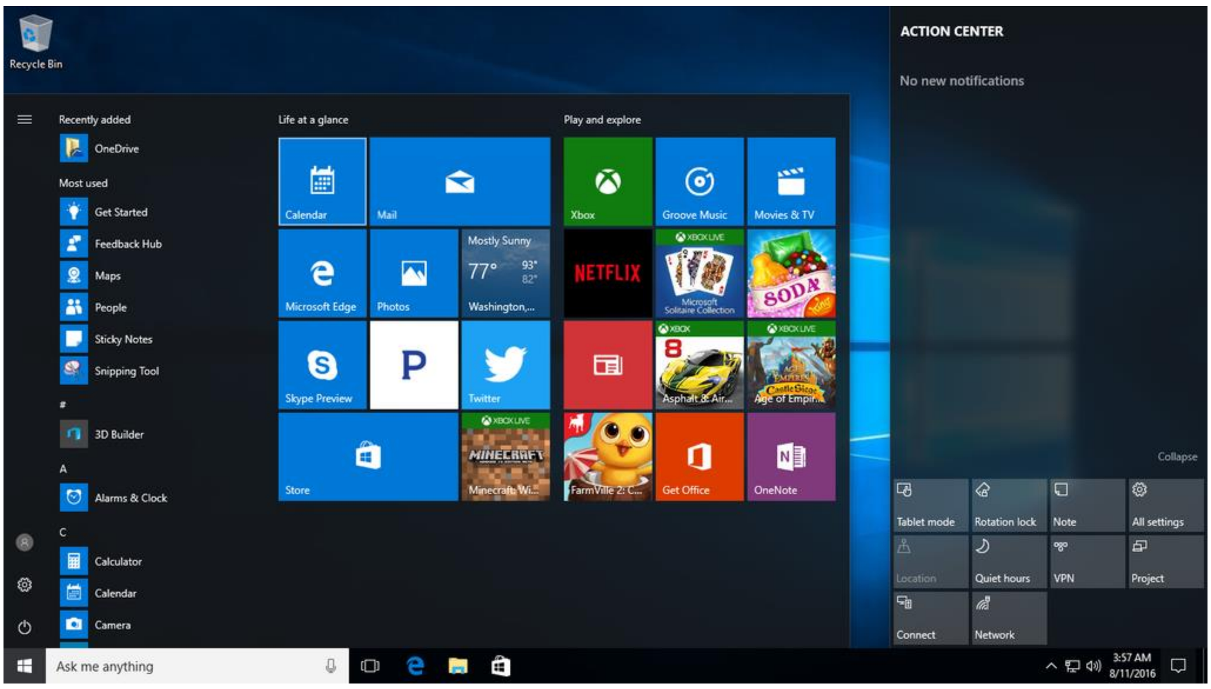
Windows 98



Windows 7

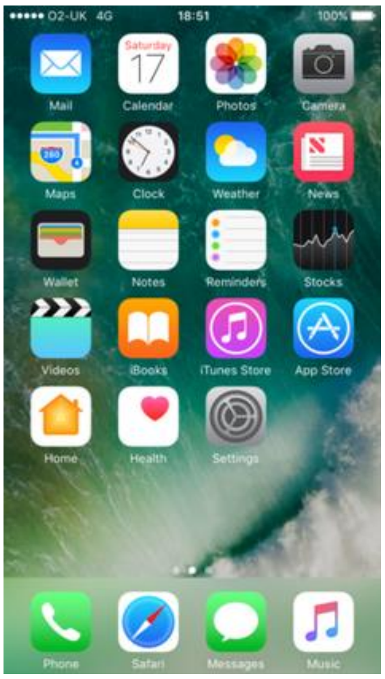


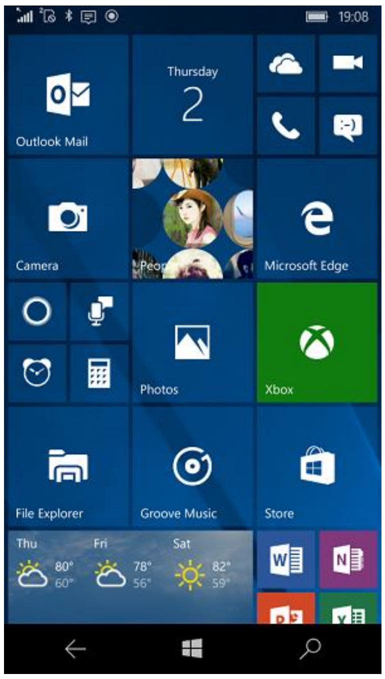
Windows 10

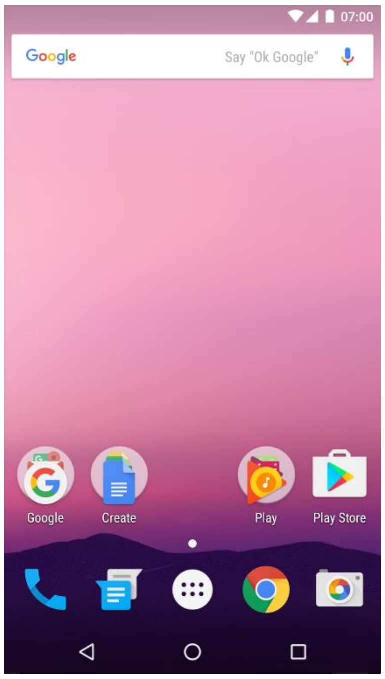


Smartphone Operating System GUI

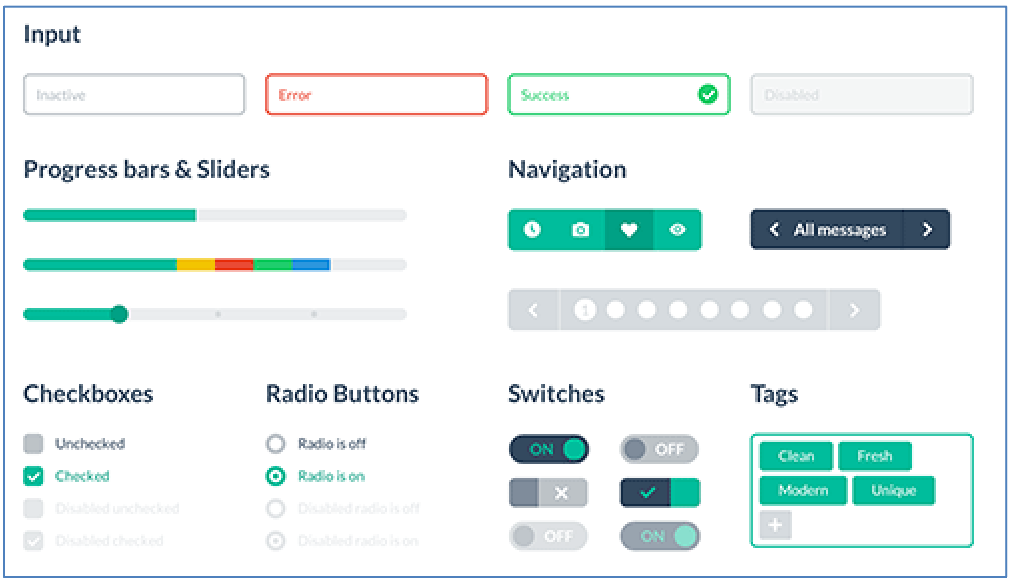
Apple IOS | **Windows 10 Phone** | **Google Android**







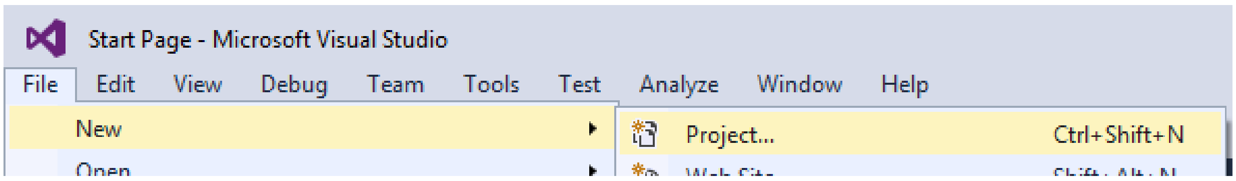
Software Applications Common UI Controls



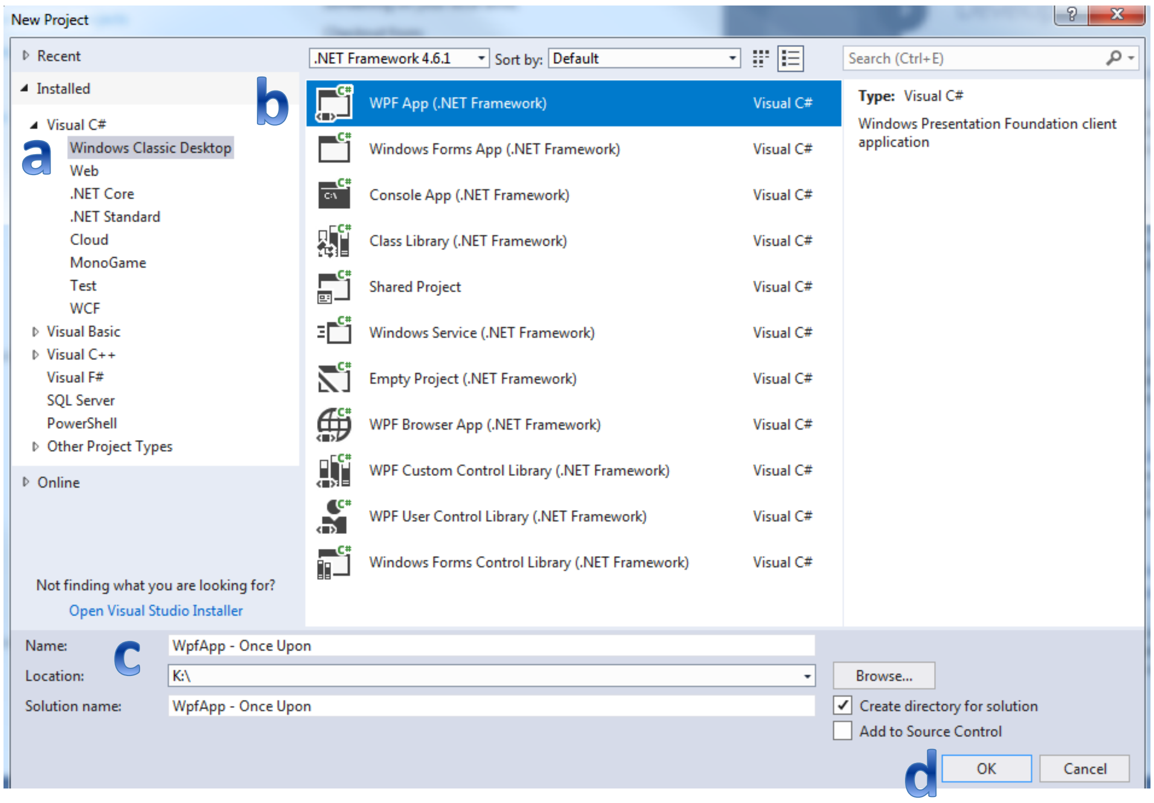
ACTIVITY: ONCE UPON A TIME APP

The best way to learn how to build a UI is to do it!

1. Create a new project



2. C# WPF application
3. A C# WPF application is a type of program and will produce a program with a window.
    a) Choose Visual C#, Windows Classic Desktop
    b) WPF App (.NET Framework)
    c) Change the name to 'WpfApp - Once Upon'
    d) Select OK to create


New Project
Recent
Installed
Visual C#
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
MonoGame
Test
WCF
Visual Basic
Visual C++
Visual F#
SQL Server
PowerShell
Other Project Types
Online
Not finding what you are looking for?
Open Visual Studio Installer
.NET Framework 4.6.1
Sort by: Default
Search (Ctrl+E)
WPF App (.NET Framework) Visual C#
Windows Forms App (.NET Framework) Visual C#
Console App (.NET Framework) Visual C#
Class Library (.NET Framework) Visual C#
Shared Project Visual C#
Windows Service (.NET Framework) Visual C#
Empty Project (.NET Framework) Visual C#
WPF Browser App (.NET Framework) Visual C#
WPF Custom Control Library (.NET Framework) Visual C#
WPF User Control Library (.NET Framework) Visual C#
Windows Forms Control Library (.NET Framework) Visual C#
Browse...
OK
Cancel
a
b
c
d

Type: Visual C#
Windows Presentation Foundation client application

Name: WpfApp - Once Upon
Location: K:\
Solution name: WpfApp - Once Upon
[✓] Create directory for solution
[ ] Add to Source Control

4. The WPF application will be created.

DEVELOPMENT STRUCTURE OF A C# WINDOWS PRESENTATION FOUNDATION (WPF)

Visual Studio will create all the necessary code for you to begin. You don't need to worry about some of the parts given to you now.

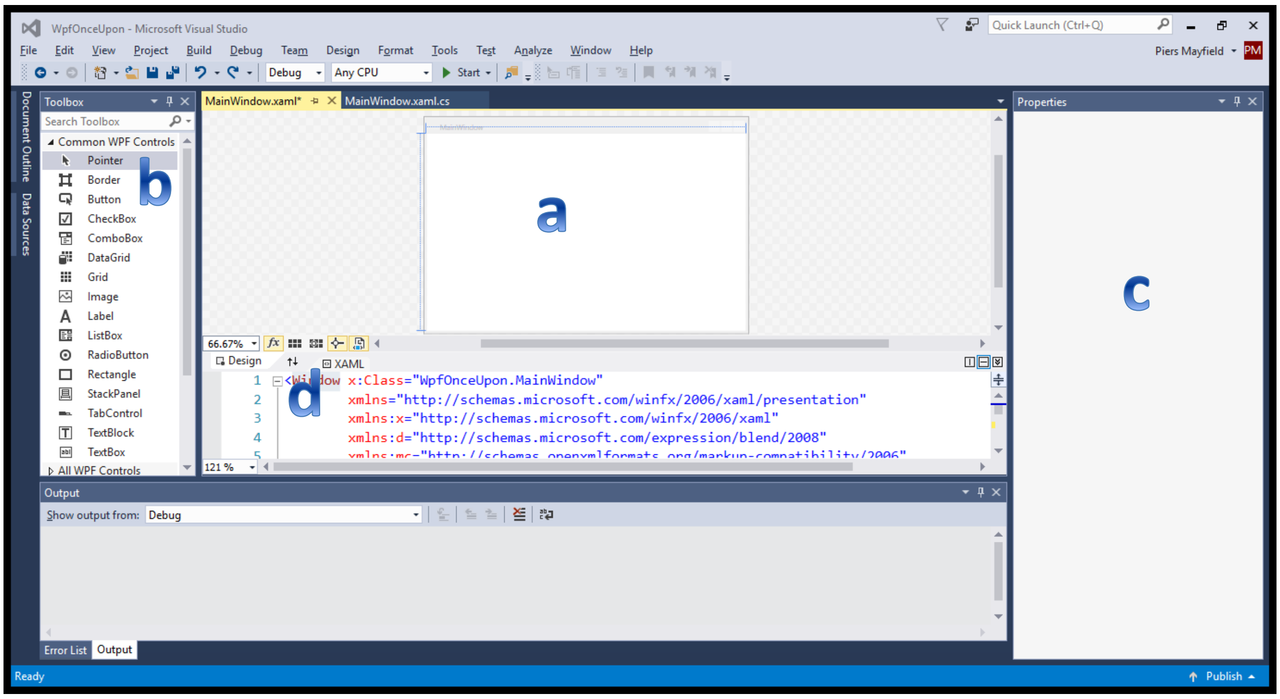
You will be working with:

a) Window designer
   - Drag and drop controls to create your windows application

b) Toolbox
   - All the controls you can use e.g. label, button, checkbox

c) Properties
   - Information about each control e.g. name, contents, size colour

d) XAML
   - This is for advanced developers, let Visual Studio handle do this for you
   - Collapse the pane for now to create more space



```
<Window x:Class="WpfOnceUpon.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

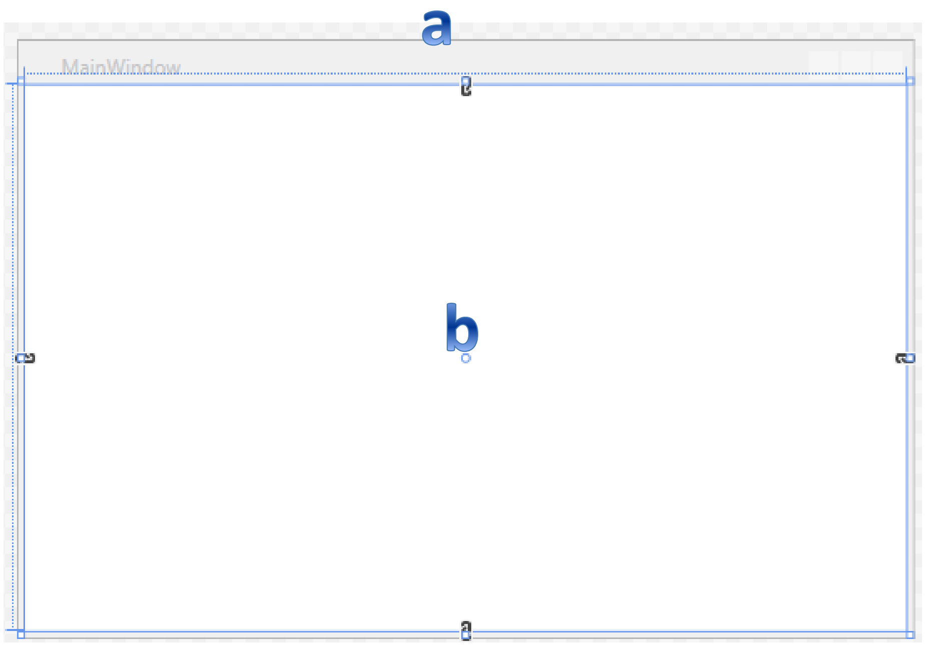
THE ANATOMY OF A WINDOW

The window created in a WPF application is made up of two main parts.

a) MainWindow
    a. The window presents and contains the content and allows the user to interact

b) Grid
    a. The grid allows you to place and organise controls in the content area of the window

    Note: If you delete the grid you cannot place controls, so be careful!

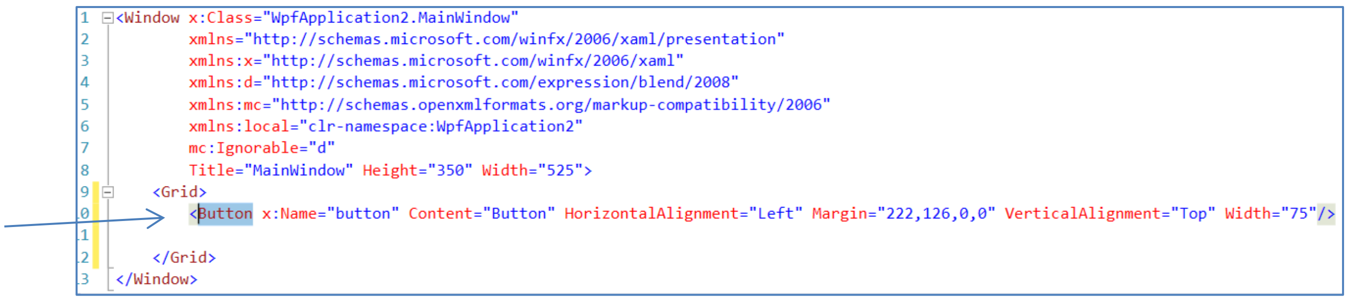


WHAT IS XAML?

XAML is similar to HTML is the respect that it can be used to define the content and behaviour of the window, grid and controls.

The XAML mark-up is created when you drag and drop controls onto the grid. If you wanted, you could type the XAML definition. Just let Visual Studio handle this for now, when you become more confident you might just type your own definitions.

Example XAML for a button:



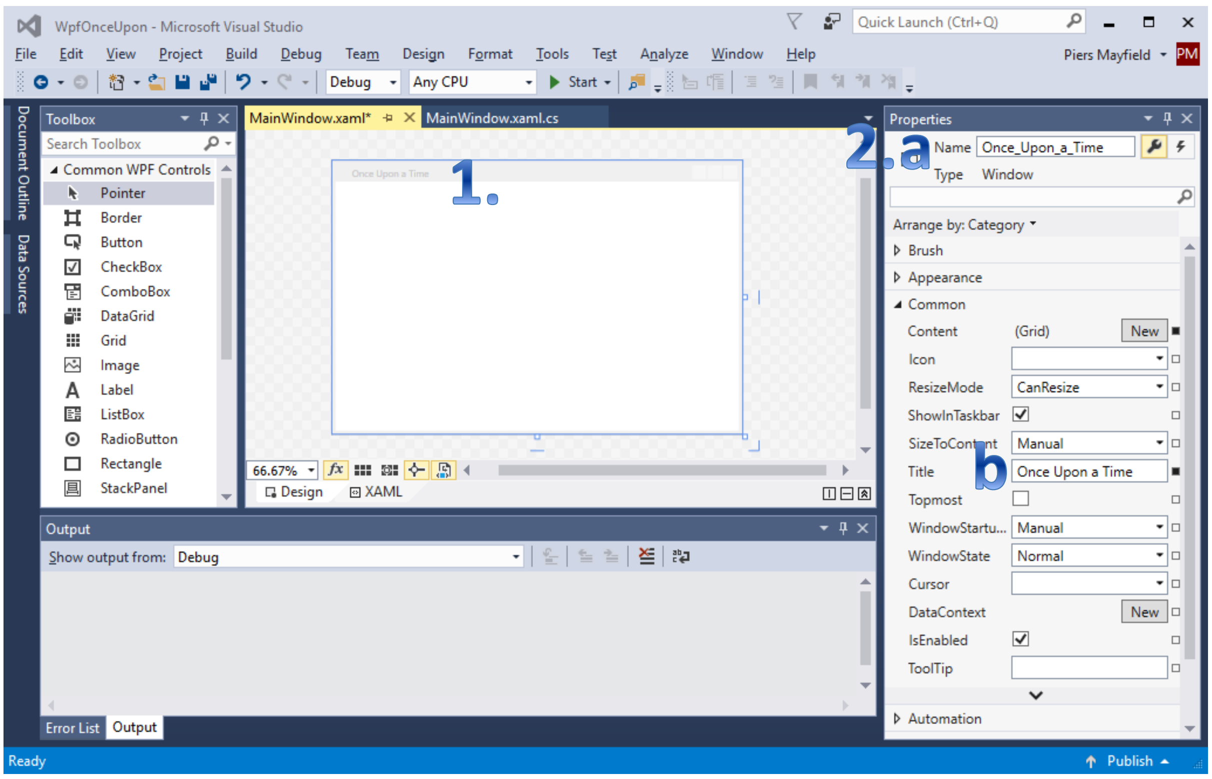
```
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfApplication2"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button x:Name="button" Content="Button" HorizontalAlignment="Left" Margin="222,126,0,0" VerticalAlignment="Top" Width="75"/>

    </Grid>
</Window>
```

SETTING UP THE MAIN WINDOW

It is always good practise to name objects, let's start with the main window.

1. Select the MainWindow, click on the grey area
2. The MainWindow properties will appear

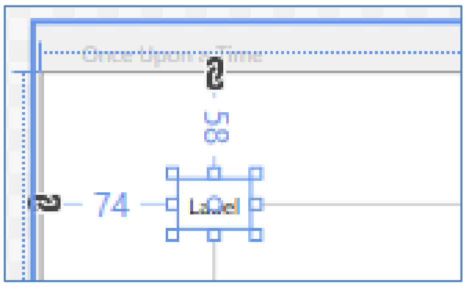
3. Update the properties
   a. **Name**:
      - Give the MainWindow a name 'Once Upon a Time'
      - This is how the program will know which window to work with
      - The window Eventually in larger applications you will have more than one window
   b. Common > **Title:**
      - Give the window a title 'Once Upon a Time'
      - This is what the user will see



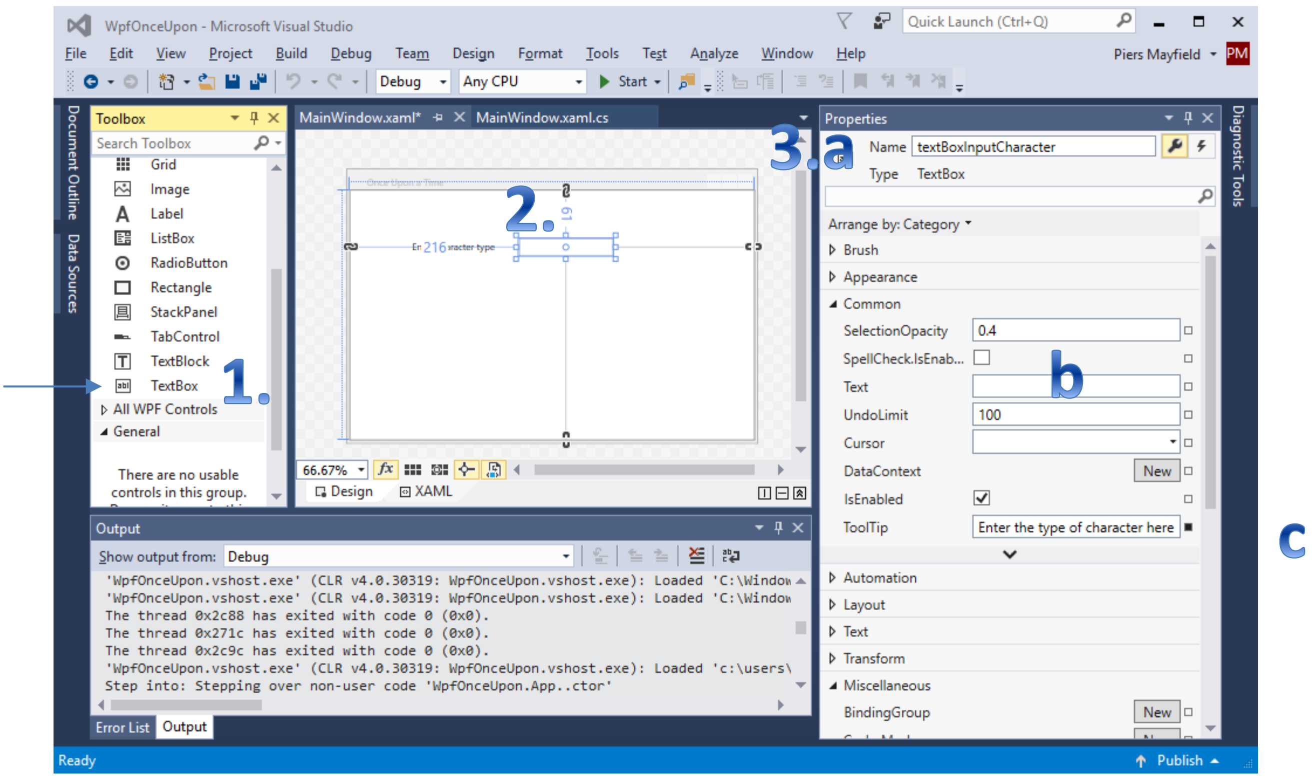
Name	Once_Upon_a_Time
Type	Window
Content	(Grid)
Icon	
ResizeMode	CanResize
ShowInTaskbar	✓
SizeToContent	Manual
Title	Once Upon a Time
Topmost	
WindowStartu...	Manual
WindowState	Normal
Cursor	
DataContext	
IsEnabled	✓
ToolTip	

ADDING A LABEL

Labels can be used to give instructions to the user and for outputting text.



1. Select label from the Toolbox
2. Drag and drop the label onto the window
3. The label properties will appear
4. Update the label properties
   a. **Name**:
      - Give the label a name ‘labelCharacter’
      - This is how the program will know which label to work with
   b. Common > **Content:**
      - Give the label the content of ‘Enter character type’
      - This is what will appear in the label
   c. Common > **ToolTip:**
      - Give the ToolTip the content of ‘giant, witch, sponge etc.’
      - This is what will popup when a user hovers over the label



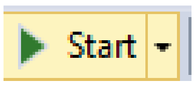
Name textBoxInputCharacter
Type TextBox

SelectionOpacity	0.4
SpellCheck.IsEnab...	☐
Text	
UndoLimit	100
Cursor	
DataContext	New
IsEnabled	✓
ToolTip	Enter the type of character here
BindingGroup	New

Output

Show output from: Debug

```
'WpfOnceUpon.vshost.exe' (CLR v4.0.30319: WpfOnceUpon.vshost.exe): Loaded 'C:\Window
'WpfOnceUpon.vshost.exe' (CLR v4.0.30319: WpfOnceUpon.vshost.exe): Loaded 'C:\Window
The thread 0x2c88 has exited with code 0 (0x0).
The thread 0x271c has exited with code 0 (0x0).
The thread 0x2c9c has exited with code 0 (0x0).
'WpfOnceUpon.vshost.exe' (CLR v4.0.30319: WpfOnceUpon.vshost.exe): Loaded 'c:\users\
Step into: Stepping over non-user code 'WpfOnceUpon.App..ctor'
```

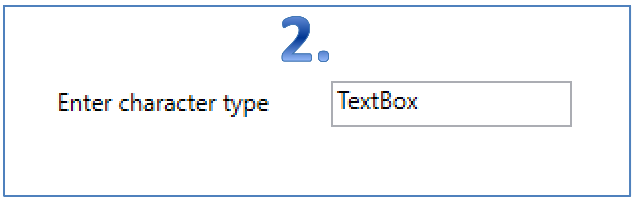
5. Press  to test what the form looks like. Close once tested.

ADDING A TEXT BOX

Text boxes are used to get an input from the user.

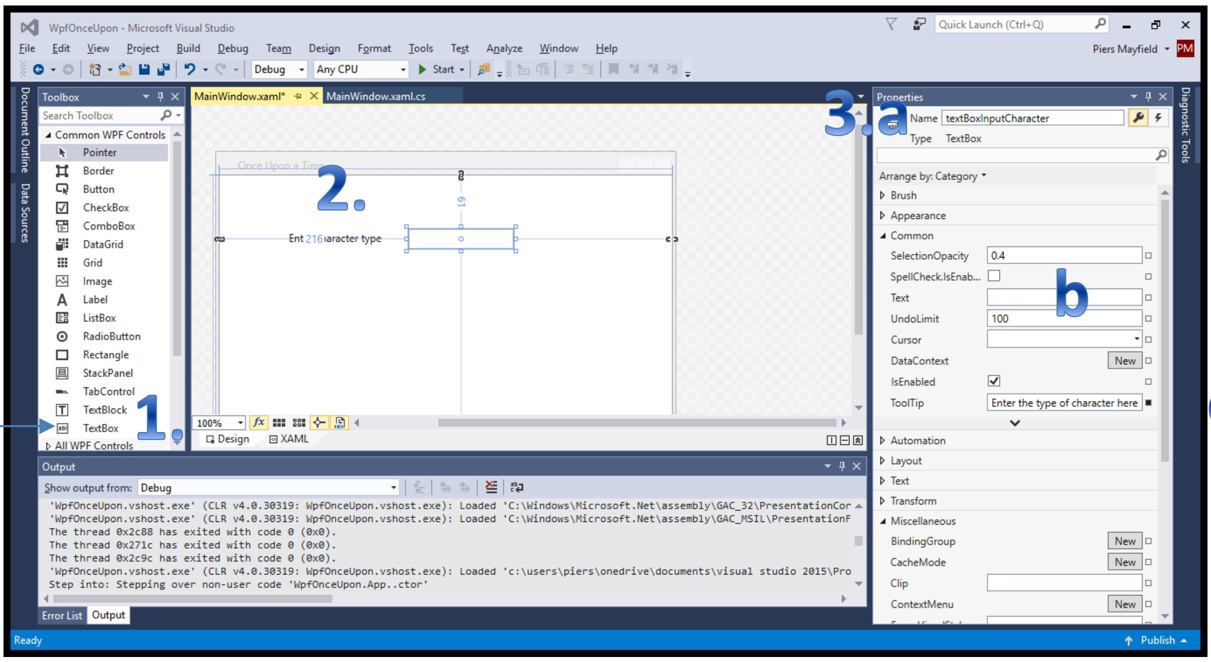
1. Select TextBox from the Toolbox

2. Drag and drop the TextBox onto the window, next to the label



3. The TextBox properties will appear

4. Update the TextBox properties
   a. **Name**:
      - Give the TextBox a name ‘textBoxInputCharacter’
      - This is how the program will know which TextBox to work with
   b. Common > **Text:**
      - ***Delete*** the TextBox text so that it is blank
      - This is where the user will type
   c. Common > **ToolTip:**
      - Give the ToolTip the content of ‘Enter the type of character here’
      - This is what will popup when a user hovers over the TextBox



Name: textBoxInputCharacter
Type: TextBox

SelectionOpacity	0.4
SpellCheck.IsEnab...	☐
Text	
UndoLimit	100
Cursor	
DataContext	New
IsEnabled	✓
ToolTip	Enter the type of character here
BindingGroup	New
CacheMode	New
Clip	
ContextMenu	New

Show output from: Debug

```
'WpfOnceUpon.vshost.exe' (CLR v4.0.30319: WpfOnceUpon.vshost.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_32\PresentationCor
'WpfOnceUpon.vshost.exe' (CLR v4.0.30319: WpfOnceUpon.vshost.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\PresentationF
The thread 0x2c88 has exited with code 0 (0x0).
The thread 0x271c has exited with code 0 (0x0).
The thread 0x2c9c has exited with code 0 (0x0).
'WpfOnceUpon.vshost.exe' (CLR v4.0.30319: WpfOnceUpon.vshost.exe): Loaded 'c:\users\piers\onedrive\documents\visual studio 2015\Pro
Step into: Stepping over non-user code 'WpfOnceUpon.App..ctor'
```

5. Press Start to test what the form looks like. Close once tested.

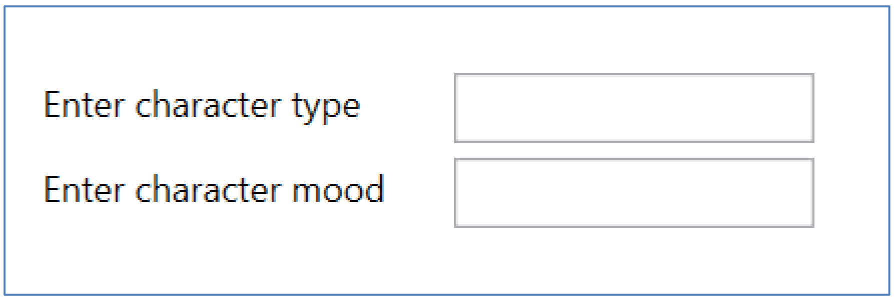
ADDING A LABEL AND TEXTBOX

Repeat the steps above to add a label and textbox for the following:

1. ***Label*** properties
   a. **Name**:
      - Give the label a name ‘labelMood’
   b. Common > **Content:**
      - Give the label the value of ‘Enter the character mood’
   c. Common > **ToolTip:**
      - Give the ToolTip the content of ‘happy, sad, grumpy, etc.’

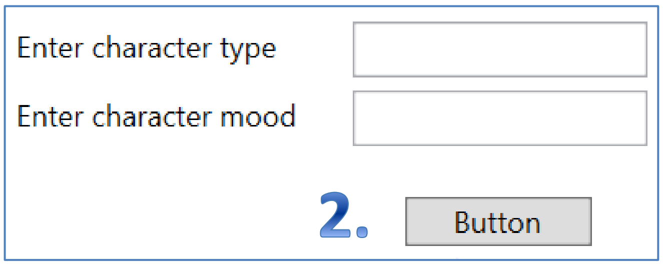
2. ***TextBox*** properties
   a. **Name**:
      - Give the TextBox a name ‘textBoxInputMood’
   b. Common > **Text:**
      - ***Delete*** the TextBox text so that it is blank
   c. Common > **ToolTip:**
      - Give the ToolTip the content of ‘Enter the character mood here’

For example:



ADDING A BUTTON

Buttons are used to trigger events, when the button is pressed the code you are going to write will be executed.



Enter character type

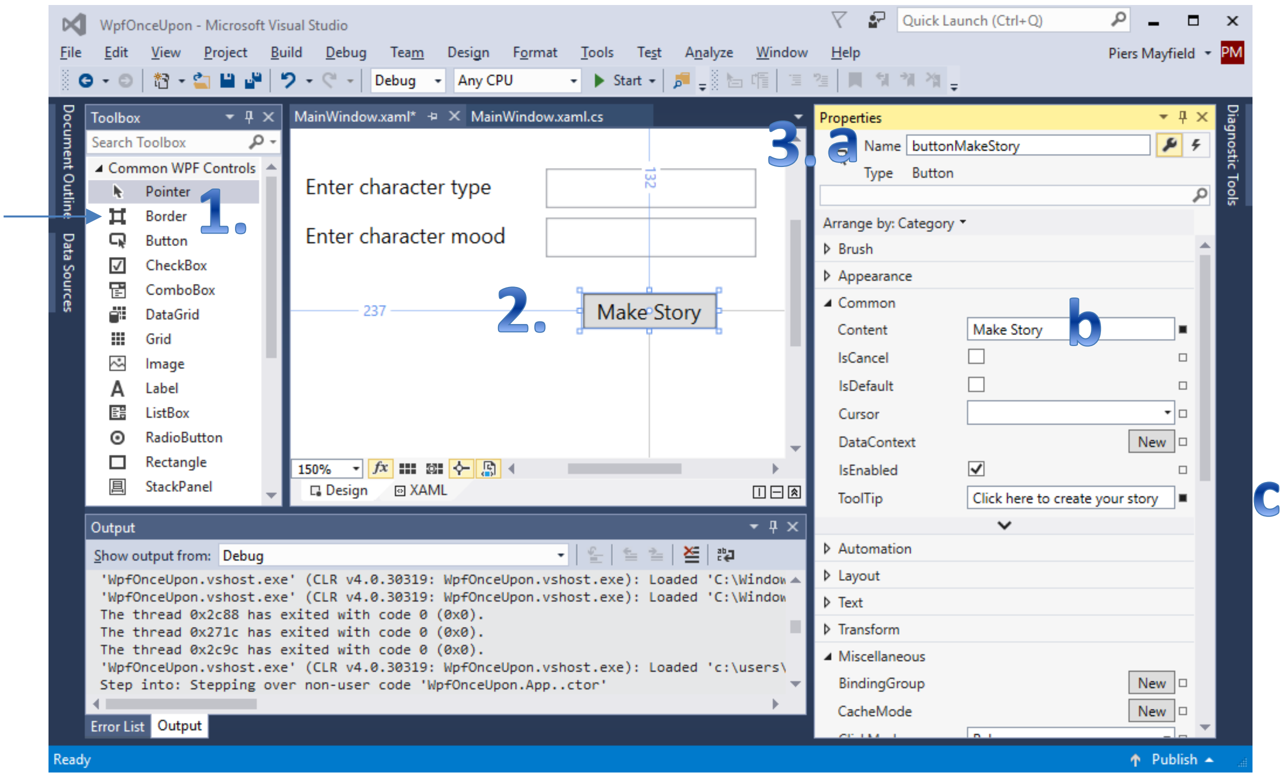
Enter character mood

1. Select TextBox from the Toolbox

2. Drag and drop the Button onto the window

3. The TextBox properties will appear

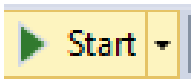
4. Update the ***Button*** properties
   a. **Name**:
      - Give the Button a name ‘buttonMakeStory’
      - This is how the program will know which button to work with and what to call the event in the code.
   b. Common > **Content:**
      - Give the Button label the value of ‘Make Story’
      - This is the button label
   c. Common > **ToolTip:**
      - Give the Button ToolTip the content of ‘Click here to create your story’
      - This is what will popup when a user hovers over the Button



Name buttonMakeStory
Type Button
Content Make Story
ToolTip Click here to create your story

Show output from: Debug

```
'WpfOnceUpon.vshost.exe' (CLR v4.0.30319: WpfOnceUpon.vshost.exe): Loaded 'C:\Window
'WpfOnceUpon.vshost.exe' (CLR v4.0.30319: WpfOnceUpon.vshost.exe): Loaded 'C:\Window
The thread 0x2c88 has exited with code 0 (0x0).
The thread 0x271c has exited with code 0 (0x0).
The thread 0x2c9c has exited with code 0 (0x0).
'WpfOnceUpon.vshost.exe' (CLR v4.0.30319: WpfOnceUpon.vshost.exe): Loaded 'c:\users\
Step into: Stepping over non-user code 'WpfOnceUpon.App..ctor'
```

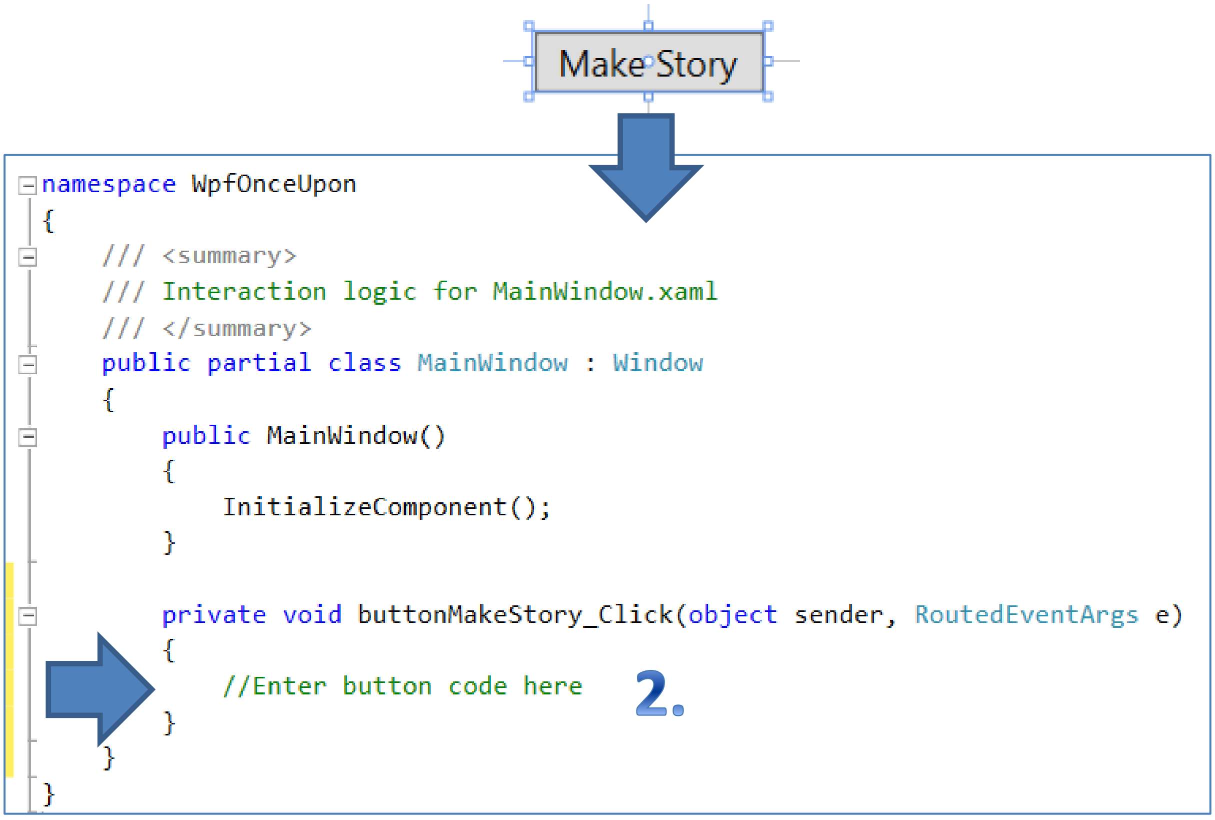
5. Press  to test what the form looks like. Close once tested.

CREATING THE BUTTON EVENT CODE

When the button is clicked, this will trigger the event that runs the code. Visual Studio will help you create the code to get you started.

The code you will create is similar to the console application that you created earlier but adapted to work with a button.

1. Double click on the button to create the button event code starting point



```
namespace WpfOnceUpon
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void buttonMakeStory_Click(object sender, RoutedEventArgs e)
        {
            //Enter button code here
        }
    }
}
```

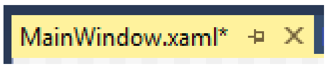
2. The code you will create will only run when the ‘Make Story’ button is clicked.

CODING CONTROL PROPERTIES - MAKING A STRING APPEAR

To get started you will need to understand a bit about how the control properties.

Each control property behaves a bit like a variable, you can store and change information held in them. BUT you must access them in a specific way, this is why each control was given a unique name.

Add a new label

1. Go to the MainWindow.xaml 

2. Drag and drop a t***extBlock*** below the button
   - Use a textBlock because the text will wrap around automatically

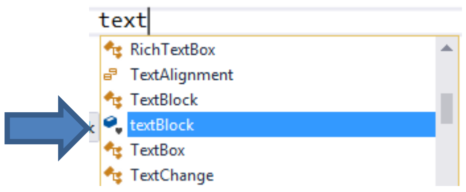
3. Update the textBlock propertiest
   a. **Name**:
      - Give the textBlock a name 'textBlockStoryOutput'

   b. Common > **Text:**
      - ***Delete*** the TextBlock the text, the label is still there but empty
      - The code you create will update the content

Adding the code

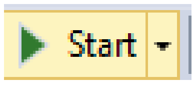
You will need to know how the textBlock is named to call it in the code.

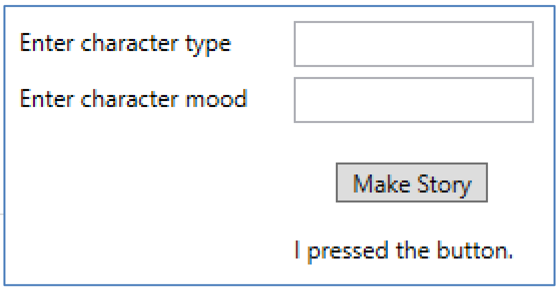
- Luckily if you followed the naming example this is easy, **intellisense** helps you:



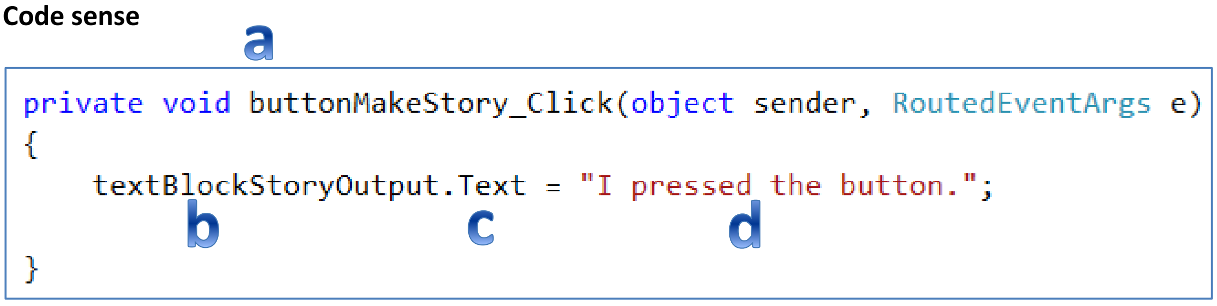
1. ***Type the following code***, or your version of it.

```
private void buttonMakeStory_Click(object sender, RoutedEventArgs e)
{
    textBlockStoryOutput.Text = "I pressed the button.";

}
```

2. Press start  and press the button to make the string appear.



Code sense



```
private void buttonMakeStory_Click(object sender, RoutedEventArgs e)
{
    textBlockStoryOutput.Text = "I pressed the button.";
}
```

a) This is the button event starting point

b) This is the name of the TextBlock control

c) This is the TextBlock control property that will given a value

d) This is the string value that will appear when the button is pressed

Improving the code

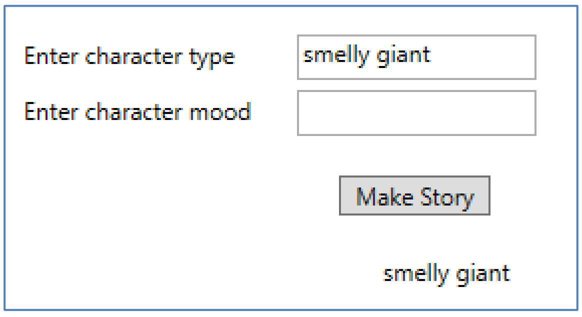
At the moment the *labelStoryOutput* has a fixed string given to it. The next step is to get the user to enter the string value.

There are different ways of doing this, you will use a variable in a very similar way to the earlier console program. This allows you more flexibility with your code.

1. ***Type/edit this code***.

```
private void buttonMakeStory_Click(object sender, RoutedEventArgs e)
{
    String character = textBoxInputCharacter.Text;
    textBlockStoryOutput.Text = character;
}
```

2. Press start Start and press the button to make the string appear.

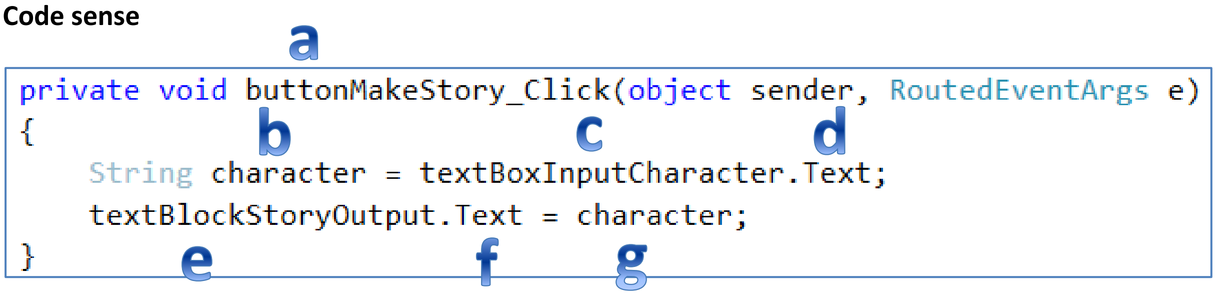


Enter character type smelly giant

Enter character mood

smelly giant

Code sense



```
private void buttonMakeStory_Click(object sender, RoutedEventArgs e)
{
    String character = textBoxInputCharacter.Text;
    textBlockStoryOutput.Text = character;
}
```

a) This is the button event starting point

b) This is the name of the string variable

c) This is the name of the textBox control to get the string value from

d) This is the property of the Textbox control that stores the value typed by the user

e) This is the button event starting point

f) This is the name of the textBlock control

g) This is the textBlock control property that will given a value

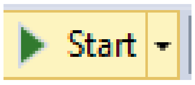
e) This is the string value that will appear when the button is pressed, in this case the variable value

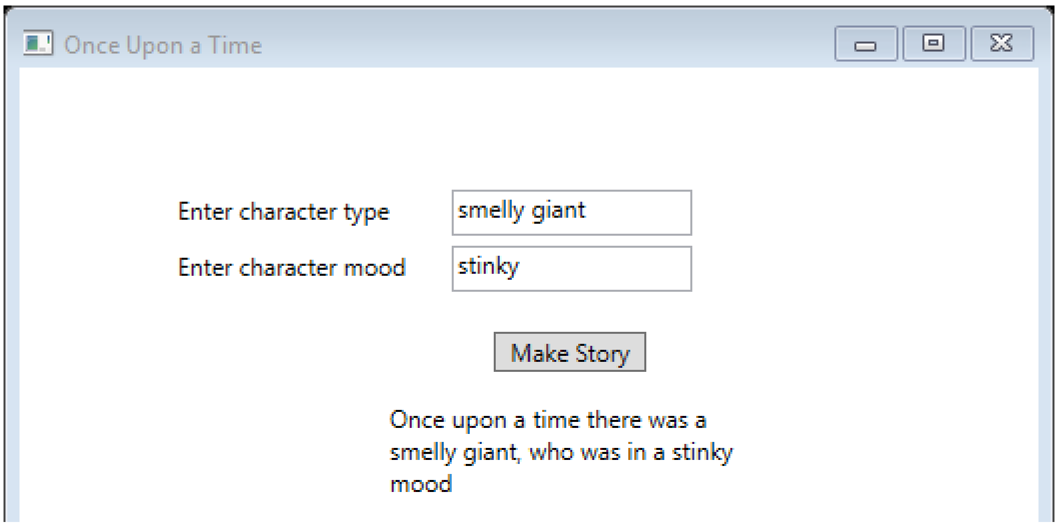
Finishing the code

Time to finish the story code.

- ***Type/edit this code***.

```
private void buttonMakeStory_Click(object sender, RoutedEventArgs e)
{
    String character = textBoxInputCharacter.Text;
    String mood = textBoxInputMood.Text;

    textBlockStoryOutput.Text = string.Format("Once upon a time there was a {0}, who was in a {1} mood"
                                                , character, mood );
}
```

- Press start  and press the button to make finished application appear.



```
Once upon a time there was a
smelly giant, who was in a stinky
mood
```

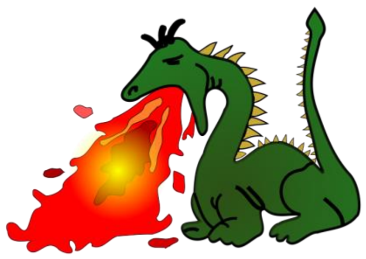
- Finally, add comments to your code.
    - The ***string.Format()*** function/method was used so that placeholders can be used. It is written on two lines to make it fit on the page.

PRACTICE ACTIVITY: WRITE A SHORT STORY UI

- Design and create a WPF UI for your short story.
- Use your short story console code as a starting point, edit the code to make it work with the WPF User Interface.
- Once finished investigate how to add images to the WPF.

  Hint: Save the image to the same folder as you project

Note: You might find it helpful to design your application on paper before you begin.



A BIT OF MATHS

Different types of numbers are used in many programs and it is important for you to know how to work with them in C#.

TYPES OF NUMBER

To be able to work with numbers you will need to recognise the most common number data types. You will typically use an **int**, **float** or **double** but other types are included.

C# stores numbers in a binary format so 42 in 8-bit binary is **00101010**; this is not obvious unless you have some experience with computer science.

Type	Description	+ and -
int	32-bit *signed* integer	Yes
float	32-bit single-precision floating point number	Yes
double	64-bit double-precision floating point number	Yes
decimal	128-bit decimal with 28 significant digits	Yes
long	64-bit *signed* integer	Yes
short	16-bit *signed* integer	Yes
byte	8-bit *unsigned* integer	No

SIGNED AND UNSIGNED INTEGERS

Both positive and negative numbers may need to be stored and this is when signed and unsigned numbers become important.

- Unsigned = only positive numbers, greater than or equal to zero
- Signed = Both positive and negative numbers

In a signed (negative) binary number, the first bit indicates whether it is a negative number.

- 42 in 8-bit binary is **00101010**
- -42 in 8-bit binary is **10101010**

BIT SIZE

The number of bits indicates the size of the number that can be stored. The more bits the larger the number or more precise, if dealing with fractional numbers.

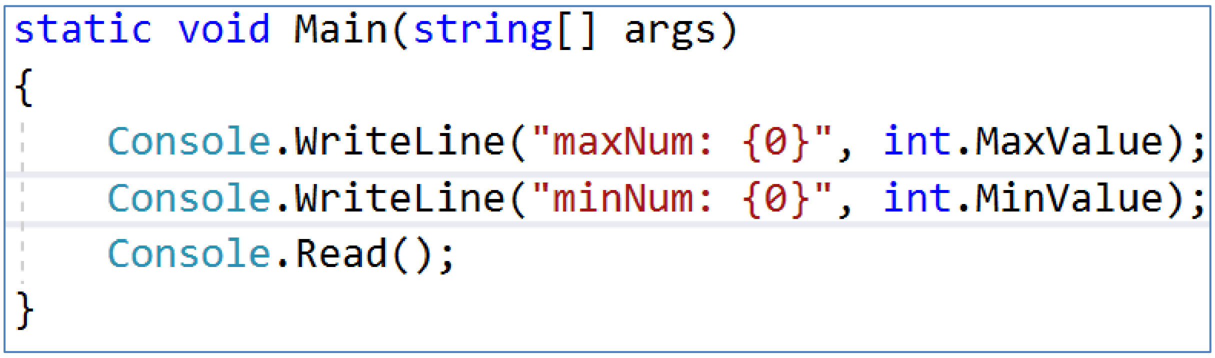
Bits	Unsigned (+)	Signed (- and +)
8	0 to 255	-128 to 127
32	0 to 4,294,967,295	-2,147,483,648 to 2,147,483,647
64	0 to 18,446,744,073,709,551,615	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Note: signed numbers, which use the most significant bit (MSB) method, use the first bit to indicate whether the number is positive of negative and not to represent a number, so have a smaller range of numbers.

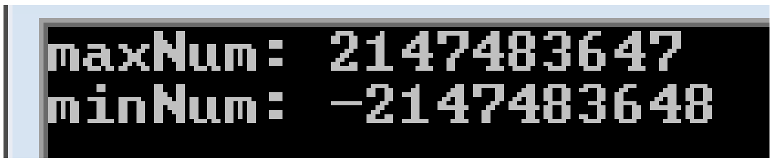
FINDING THE RANGE OF NUMBERS

Luckily C# provides you with a way to find the range of numbers. You never know, when you might need to!

Use the **numberType.MaxValue** or **numberType.MinValue**, for example:



```
static void Main(string[] args)
{
    Console.WriteLine("maxNum: {0}", int.MaxValue);
    Console.WriteLine("minNum: {0}", int.MinValue);
    Console.Read();
}
```



```
maxNum: 2147483647
minNum: -2147483648
```

OPERATORS

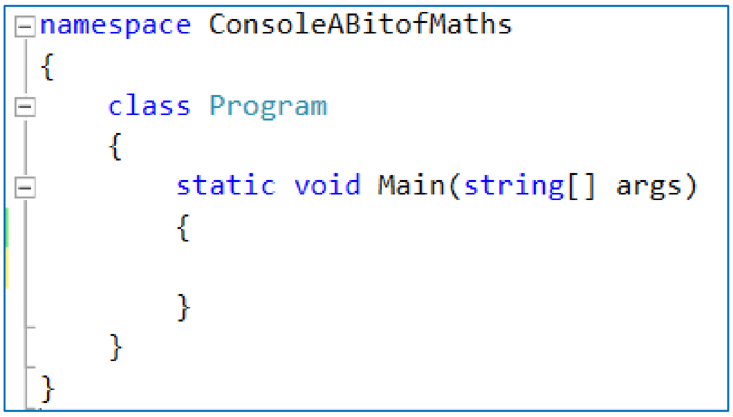
C# can perform any calculation you need but before you try you will need to know what operators to user.

Mathematical Operation	Maths Operator	Maths example	C# Operator	C# Example
Addition	+	4 + 4 = 8	+	4 + 4 = 8
Subtraction	-	4 - 4 = 0	-	8 - 4 = 4
Multiplication	*	4 * 2 = 8	*	4 * 2 = 8
To the power of	X^y	8^3 (8*8*8)= 512	None	Math.Pow(8,3) = 512
Division	/	8 / 2 = 4 5 / 2 = 2.5	/	8 / 2 = 4 5 / 2 = 2.5
Modulus (remainder)	%	8 % 2 = 0 5 % 2 = 1	%	8 % 2 = 0 5 % 2 = 1

SIMPLE CALCULATIONS

Let's do some simple calculations in C#.

Create a C# console application called ***ConsoleABitofMaths***.



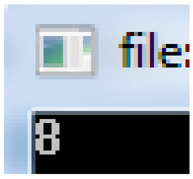
```
namespace ConsoleABitofMaths
{
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Try the code for each section

ADDITION

Addition with an int

```
static void Main(string[] args)
{
    int addInt = 4 + 4;

    Console.WriteLine(addInt);
    Console.Read();
}
```

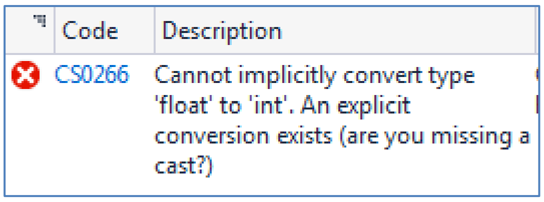


```
8
```

Adding a float using an int

```
static void Main(string[] args)
{
    int addInt = 4 + 4.5f;

    Console.WriteLine(addInt);
    Console.Read();
}
```

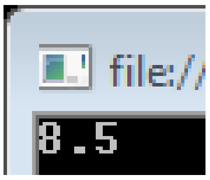
Will cause a type error



	Code	Description
❌	CS0266	Cannot implicitly convert type 'float' to 'int'. An explicit conversion exists (are you missing a cast?)

Adding a float using a float

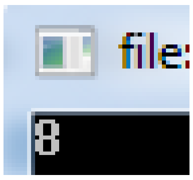
```
static void Main(string[] args)
{
    float addFloat = 4 + 4.5f;

    Console.WriteLine(addFloat);
    Console.Read();
}
```



```
8.5
```

Adding an int using a float

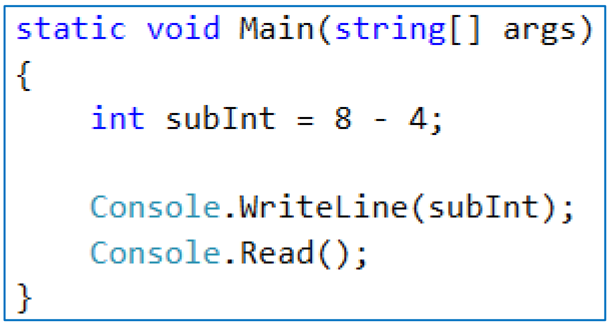
```
static void Main(string[] args)
{
    float addFloat = 4 + 4f;

    Console.WriteLine(addFloat);
    Console.Read();
}
```



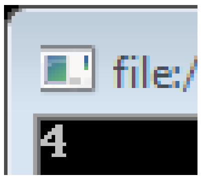
```
8
```

SUBTRACTION

Subtraction with an int

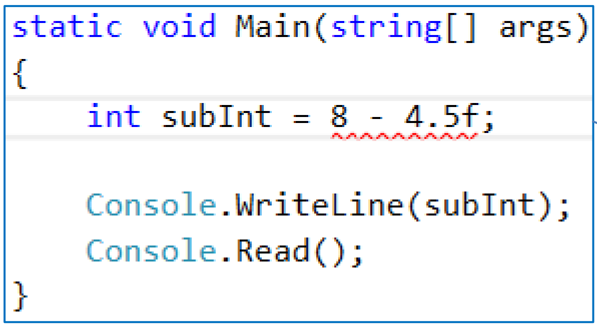


```
static void Main(string[] args)
{
    int subInt = 8 - 4;

    Console.WriteLine(subInt);
    Console.Read();
}
```



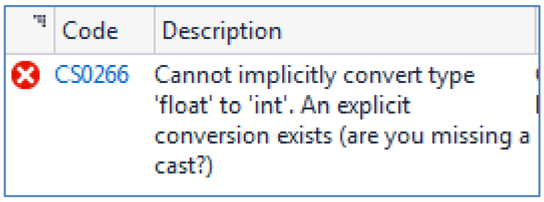
```
4
```

Subtracting a float using an int



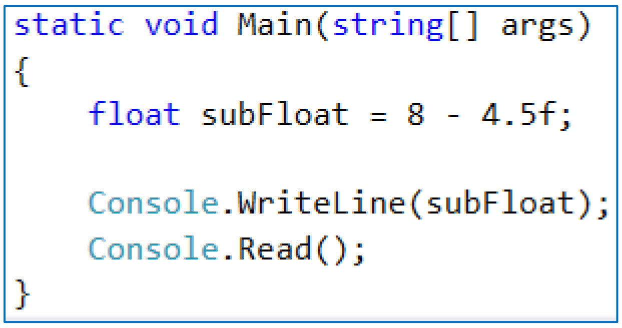
```
static void Main(string[] args)
{
    int subInt = 8 - 4.5f;

    Console.WriteLine(subInt);
    Console.Read();
}
```

Will cause an type error

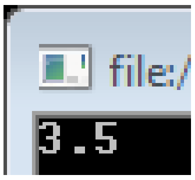


	Code	Description
⊗	CS0266	Cannot implicitly convert type 'float' to 'int'. An explicit conversion exists (are you missing a cast?)

Subtracting a float using a float



```
static void Main(string[] args)
{
    float subFloat = 8 - 4.5f;

    Console.WriteLine(subFloat);
    Console.Read();
}
```

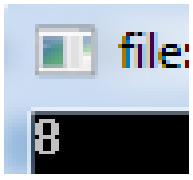


```
3.5
```

MULTIPLICATION

Multiplication with an int

```
static void Main(string[] args)
{
    int multInt = 4 * 2;

    Console.WriteLine(multInt);
    Console.Read();
}
```

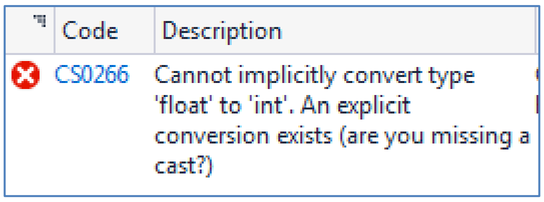


```
8
```

Multiplying with a float using an int

```
static void Main(string[] args)
{
    int multInt = 4 * 2.5f;

    Console.WriteLine(multInt);
    Console.Read();
}
```

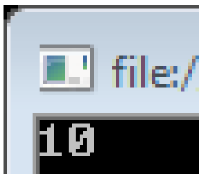
Will cause a type error



Code	Description
CS0266	Cannot implicitly convert type 'float' to 'int'. An explicit conversion exists (are you missing a cast?)

Multiplying with a float using a float

```
static void Main(string[] args)
{
    float multFloat = 4 * 2.5f;

    Console.WriteLine(multFloat);
    Console.Read();
}
```

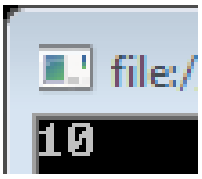


```
10
```

Multiplying with a double

A double datatype is much more forgiving than a float. You can use either an **int** or **float** value without saying which and it will just work!

```
static void Main(string[] args)
{
    double multDub = 4 * 4.5;

    Console.WriteLine(multDub);
    Console.Read();
}
```



```
10
```

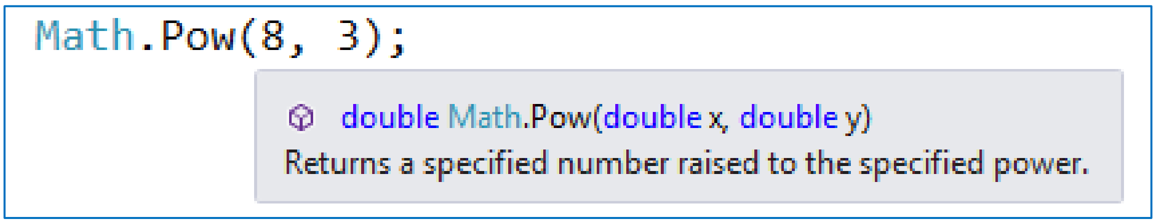
CONCLUSION

- If you are expecting result with a **float** type of number, use a **float** datatype.
- **If you are not sure if you might get a result with a fraction, use a** double **datatype to be safe.**
- If you are always working with whole numbers (integers) then use an **int** datatype.

TO THE POWER OF

The **exponent** of a number says how many times to use the number in a multiplication.

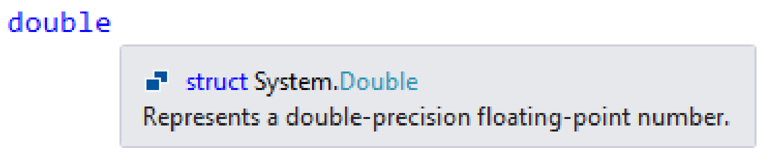
6^3 is the same as writing 6*6*, using the exponent of 3 written as 6^3 makes this easier to write.



```
Math.Pow(8, 3);
```

double Math.Pow(double x, double y)
Returns a specified number raised to the specified power.

As can be seen, C# tells us that Math.Pow() produces a datatype of **double**. This is like a **float** but with double the accuracy (precision).



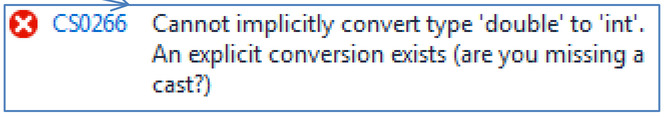
```
double
```

struct System.Double
Represents a double-precision floating-point number.

Int variable and exponent

```
static void Main(string[] args)
{
    int iHaveThePower = Math.Pow(8, 3);
    Console.WriteLine(iHaveThePower);

    Console.Read();
}
```

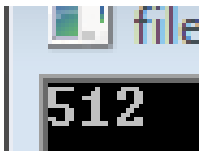
Will cause a type error, as an **int** is not a **double**



CS0266 Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)

Double variable and int or float exponent

```
static void Main(string[] args)
{
    double iHaveThePower = Math.Pow(8, 3);

    Console.WriteLine(iHaveThePower);

    Console.Read();
}
```

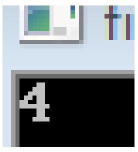


```
512
```

DIVISION

Division of an int

```
static void Main(string[] args)
{
    int divInt = 8 / 2;

    Console.WriteLine(divInt);
    Console.Read();
}
```

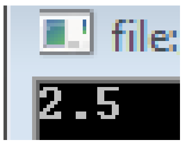


```
4
```

Dividing with a float result

```
static void Main(string[] args)
{
    float divFloat = 5 / 2f;

    Console.WriteLine(divFloat);
    Console.Read();
}
```

You must specify the divisor to be a float datatype to get a float result

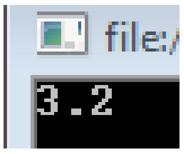


```
2.5
```

Dividing with a double result

```
static void Main(string[] args)
{
    double divDub = 8 / 2.5;

    Console.WriteLine(divDub);
    Console.Read();
}
```

There is no requirement to identify the divisor type. This makes **double** more flexible that using **float**.



```
3.2
```

Float or Double

- If you know you are always going to be given a float value then use a **float**.
- If you could be given a float or integer value use a **double**.

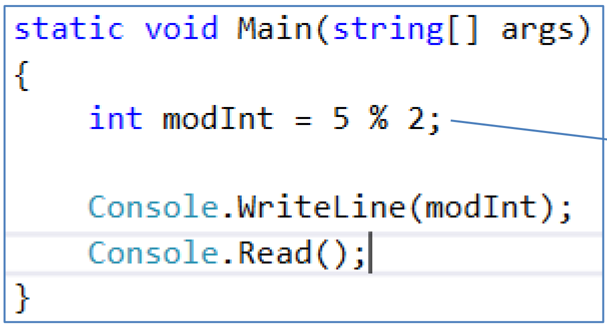
MODULO

Modulo is used to find the remainder of a division operation.

Using the modulus operator

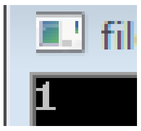
5 % 2 = 1 2 goes into 5 twice but leaves **1 remaining**, this is modulus, finding the remainder.

Modulo of an int



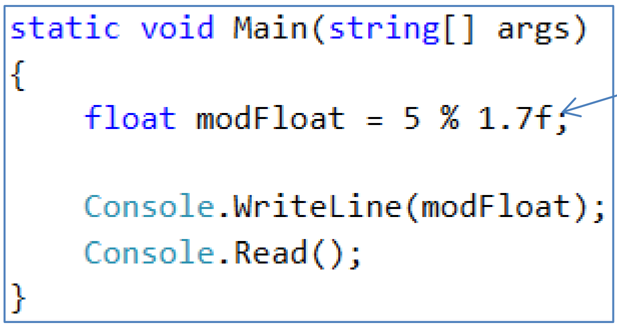
```
static void Main(string[] args)
{
    int modInt = 5 % 2;

    Console.WriteLine(modInt);
    Console.Read();
}
```

The remainder is



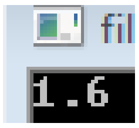
```
1
```

Modulo with a float



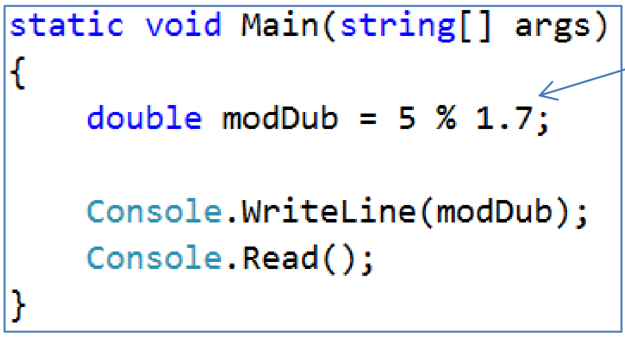
```
static void Main(string[] args)
{
    float modFloat = 5 % 1.7f;

    Console.WriteLine(modFloat);
    Console.Read();
}
```

You must specify the modulo to be a float datatype to get a float result



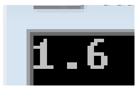
```
1.6
```

Modulo with a double



```
static void Main(string[] args)
{
    double modDub = 5 % 1.7;

    Console.WriteLine(modDub);
    Console.Read();
}
```

NO NEED to specify the datatype



```
1.6
```

RANDOMNESS

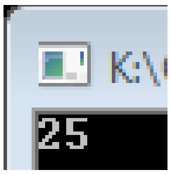
It is quite common in programs to require random numbers to be generated. Luckily, C# has a built-in class for doing that.

Remember using the string.Contains() method, random works in a similar way. Just treat it the same way for now!

A class is a bit more complex than a simple built-in method, it will create something called an object, the object is then used by the .method.

```
// Create new Random object called randNum
Random randNum = new Random();

//Use the randNum object and .Next to
//generate a random numberbetween 1 and 100
Console.WriteLine(randNum.Next(1, 100));

Console.Read();
```

Random number generated



```
25
```

USER INPUT

So far you have not been getting user input, just using C# as a calculator. As you know user input is a string, this is a problem.

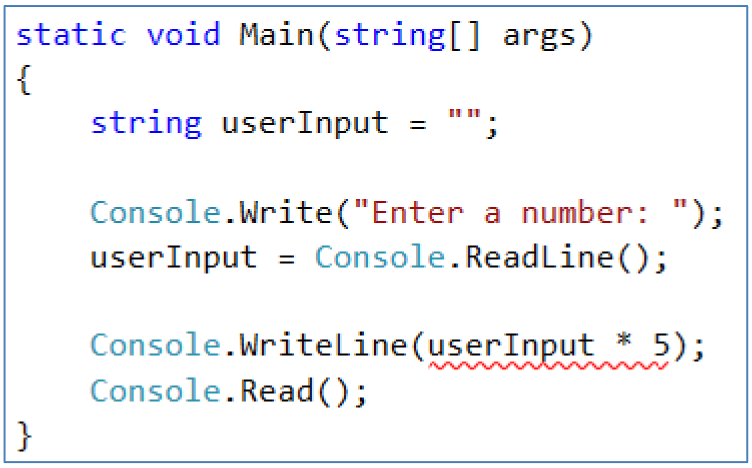
C# is very particular about data types and you can't put a string value into an **int** or **float** variable.

PARSE THE STRING

Parsing is when an item is analysed to identify (its parts) what it is, so when parsing a **string** it is first checked to see if it can be changed into another datatype.

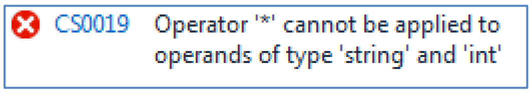
It is common to want an integer input and then do something with it, like multiply.

String errors



```
static void Main(string[] args)
{
    string userInput = "";

    Console.Write("Enter a number: ");
    userInput = Console.ReadLine();

    Console.WriteLine(userInput * 5);
    Console.Read();
}
```

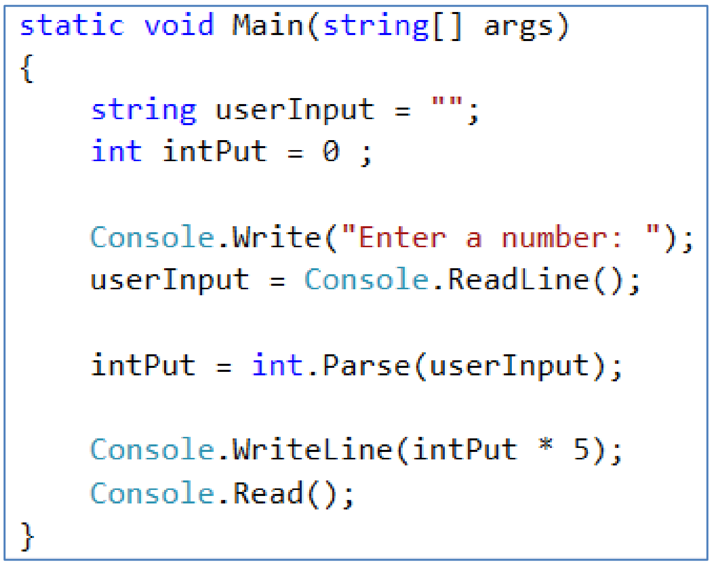
Will cause a type error



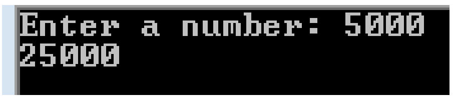
CS0019 Operator '*' cannot be applied to operands of type 'string' and 'int'

String casting

Changing the datatype is also called casting in programming but in C#, parsing is the technique used.



```
static void Main(string[] args)
{
    string userInput = "";
    int intPut = 0 ;

    Console.Write("Enter a number: ");
    userInput = Console.ReadLine();

    intPut = int.Parse(userInput);

    Console.WriteLine(intPut * 5);
    Console.Read();
}
```



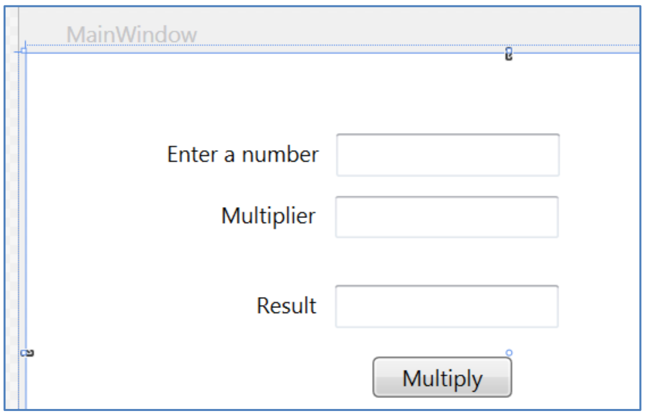
```
Enter a number: 5000
25000
```

USER INTERFACE AND CASTING

When using WPF the textBox input provides a string, so the datatype needs to be changed if it is required to behave like a number.

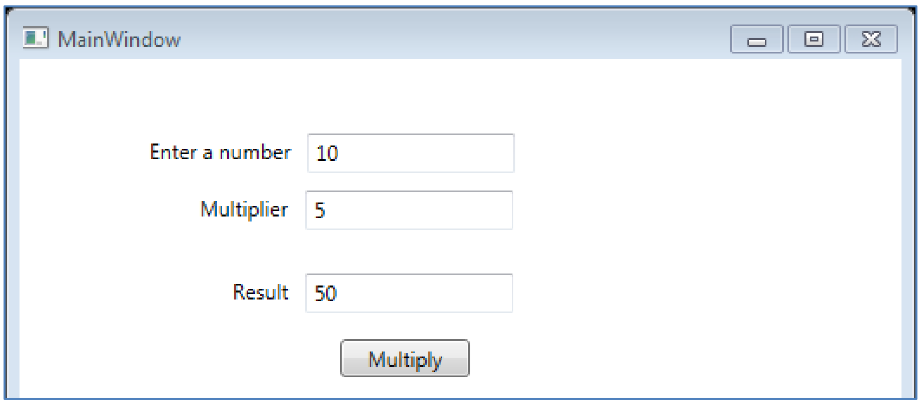
The following example multiplies two numbers, converts the textBox to an **integer**, performs a calculation and then converts the result back to a **string** to be displayed in a textBox.

User Interface



Button code

```
private void button_Click(object sender, RoutedEventArgs e)
{
    //Get the user input - parse from string to int
    int numInput = int.Parse(textBoxNum.Text);
    int numMultiplier = int.Parse(textBoxMultiplier.Text);

    //Calculate result
    int result = numInput * numMultiplier;

    //Output result - cast result integer variable to a string
    textBoxMutiplyResult.Text = result.ToString();

}
```



Enter a number 10

Multiplier 5

Result 50

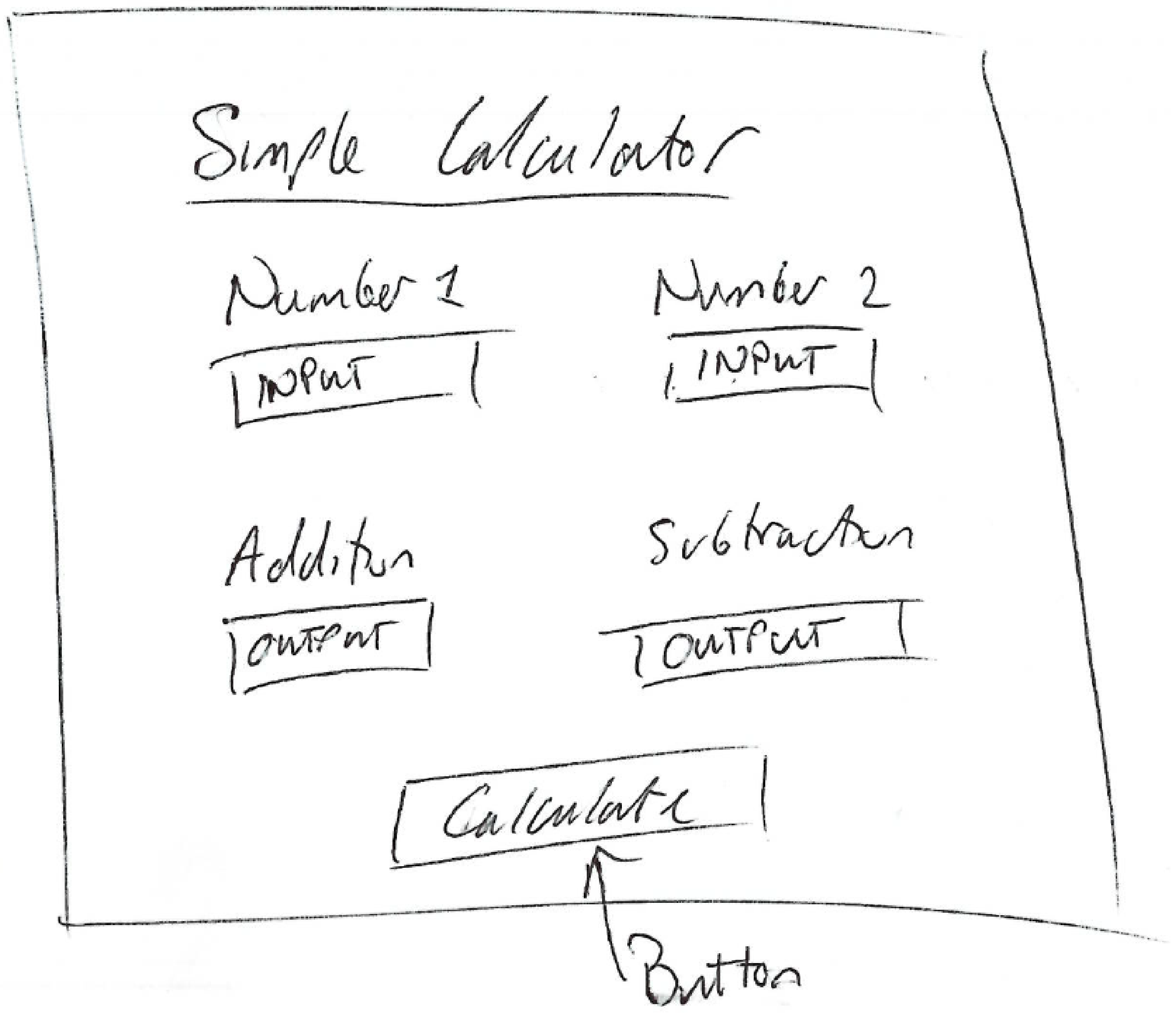
ACTIVITY: SIMPLE INTEGER CALCULATOR UI

In this activity you will build a simple calculator UI that takes in two numbers and performs different calculations.

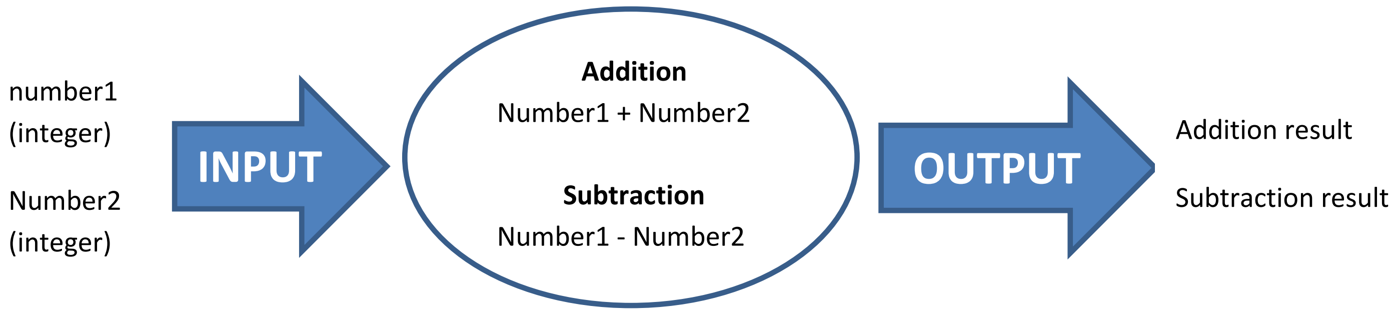
You will need to look at the prior sections and apply what you have learnt to the task.

USER INTERFACE (UI) DESIGN

It is always good to have a bit of a think about the design of you user interface. Just a quick sketch will do.



I-P-O



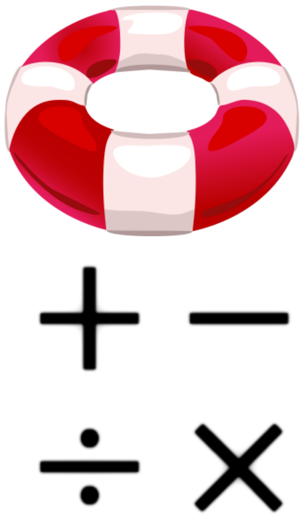
PSEUDO CODE

Please note the pseudocode doesn't try to explain how the user interface will work just the main steps required for the code.

It is up to you as the developer to work out how to create the UI and make the code work.

```
#Get user input
firstNumber ← USERINPUT
firstNumber ← USERINPUT

#Process results

addResult = firstNumber + secondNumber

subResult = firstNumber - secondNumber


#Output results

OUTPUT "addResult"

OUTPUT "subResult"
```

Activity: Simple Double Calculator



Save a copy of the simple integer calculator project and change the input and output variables to be a double.

Try using numbers with a decimal and see what happens.

Remember to:

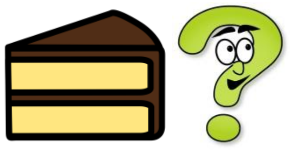
- Create an I-P-O diagram
- Design UI
- Write Pseudocode

Activity: Speed of Light Calculator

Your next task is to create a speed of light calculator. The user will input the number of seconds they want to travel for and the application will calculate distance travelled. You will need to investigate the “speed of light per second”.

Remember to:

- Create an I-P-O diagram
- Design UI
- Write Pseudocode



SELECTION

Decision making is when you make a selection, like which cake to eat; chocolate or coffee. When making a decision a criteria or condition is evaluated to help make a choice which option to choose.

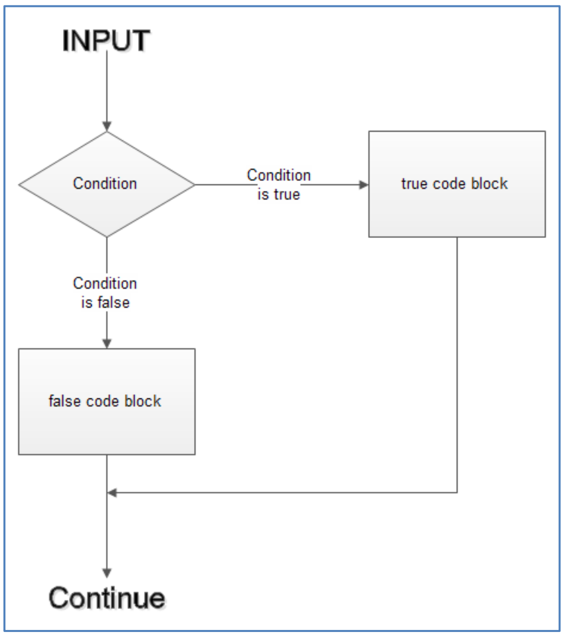
IF, ELSE

When programming you often want different code to be run depending on some criteria or condition. If the condition is met, it is true and the **true** code block runs, otherwise (else) the condition is false and the **false** code block runs. This is called branching.

PSEUDOCODE / FLOWCHART

Here is a more formal way of writing an IF, ELSE selection statement.

```
IF condition is true THEN:
    Do true code block
ELSE:
    Do false code block
```



CAKE EXAMPLE:

Criteria / Condition: No cake with coffee in it

So, written in a more formal style:

```
IF Cake EQUAL TO coffee THEN:
    Don't eat the cake
ELSE:
    Eat the cake
```

C# IF, ELSE STRUCTURE

```
if (condition)
    {
        true code block ;
    }
else
    {
        false code block ;
    }
```

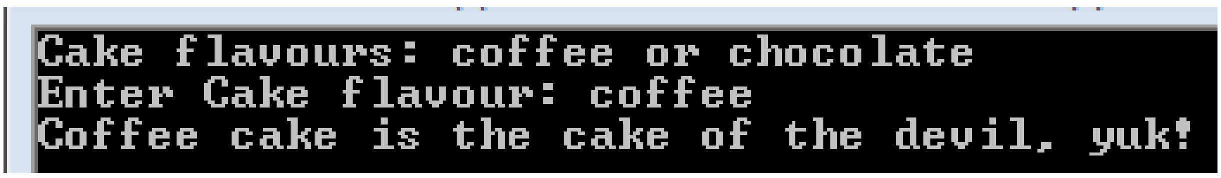
Condition in ( ) brackets

Code blocks in-between { } brackets.

Complete each code statement with a ;

A C# CONSOLE EXAMPLE:

```
static void Main(string[] args)
{
    Console.WriteLine("Cake flavours: coffee or chocolate");
    Console.Write("Enter Cake flavour: ");
    string cakeFlavour = Console.ReadLine();

    if (cakeFlavour == "coffee")
    {
        Console.WriteLine("Coffee cake is the cake of the devil, yuk!");
    }
    else
    {
        Console.WriteLine("Mmmm, yummy cake.");
    }

    Console.Read();
}
```



```
Cake flavours: coffee or chocolate
Enter Cake flavour: coffee
Coffee cake is the cake of the devil, yuk!
```

CRITERIA LOGIC

There are different ways to use logic when defining a criteria and this depends on the type of problem. For example the criteria is **"No cake with coffee in it"**.

Option 1: If the cake is coffee flavoured do not eat it

Option 2: Eat any cake that is NOT coffee flavoured

Surely they behave the same you are thinking. Let's investigate…

INPUT Flavour	Option 1	True/False	OUTPUT
Coffee	**Equal to coffee**	**True**	**Don't eat cake**
Vanilla	**Equal to coffee**	**False**	**Eat cake**
Chocolate	**Equal to coffee**	**False**	**Eat cake**
Banana	**Equal to coffee**	**False**	**Eat cake**
Coconut	**Equal to coffee**	**False**	**Eat cake**
Lemon	**Equal to coffee**	**False**	**Eat cake**

If coffee is the flavour chosen the **true** block will execute. Any other flavour and the **false** block will be run.

INPUT Flavour	Option 2	True/False	OUTPUT
Coffee	**NOT coffee**	**False**	**Don't eat cake**
Vanilla	**NOT coffee**	**True**	**Eat cake**
Chocolate	**NOT coffee**	**True**	**Eat cake**
Banana	**NOT coffee**	**True**	**Eat cake**
Coconut	**NOT coffee**	**True**	**Eat cake**
Lemon	**NOT coffee**	**True**	**Eat cake**

If coffee is the flavour chosen the **false** block will execute. Any other flavour and the **true** block will be run.

So, the same result two different ways. The only flavour you know was chosen is coffee.

What if you wanted to know which flavour was chosen?

- Well, using a simple IF, ELSE statement you can only check for one flavour.
- This is when a CASE statement becomes useful.

CASE STATEMENT

A CASE statement allows a variable to be compared to a range of values, whereas an IF, ELSE can only check one. Each value compared is called a case, hence CASE statement and the variable being compared is known as a switch. This is why it is called a ***SWITCH*** statement in C#.

Advantages of CASE/Switch

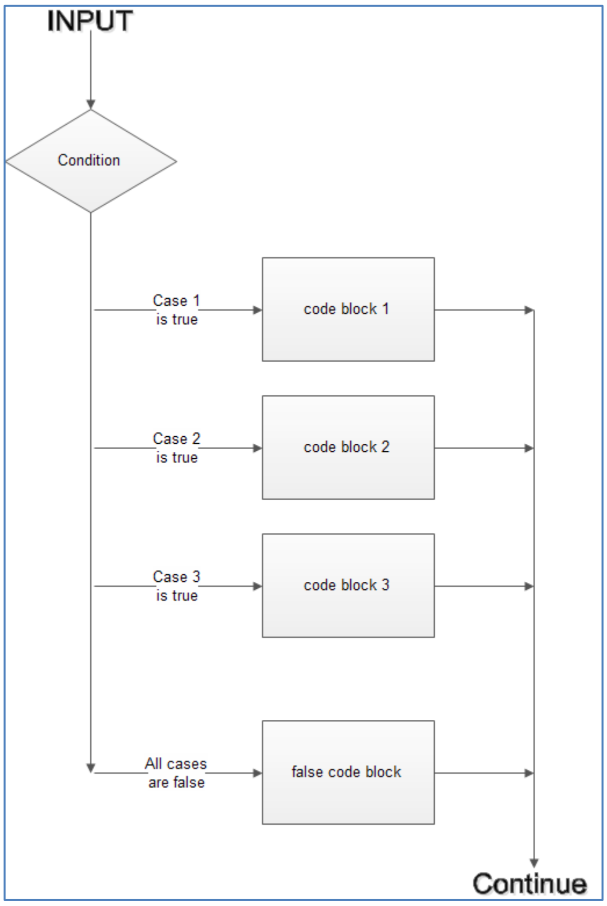
- Easier to understand, and therefore easier to maintain
- Easier to debug and check is working correctly for each case

Using this technique a programmer can definitively know when more options have been chosen.

PSEUDOCODE / FLOWCHART

Here is a more formal way of writing an IF, ELSE selection statement.

```
Var ← USERINPUT

CASE Var OF

    Case 1:    Case 1 is true
               do code block

    Case 2:    Case 2 is true

               do code block

    Case 3:    Case 3 is true

               do code block

ELSE

    All Cases are false, do code block
```



CAKE EXAMPLE:

Criteria / Condition: No cake with coffee in it

```
    cakeFlavour ← USERINPUT

    CASE cakeFlavour OF

        Coffee:         OUTPUT "OK, if you must"

        Vanilla:        OUTPUT "Excellent choice"

        Chocolate:      OUTPUT "Very tasty"

        Lemon:          OUTPUT "A connoisseur of cake"

    ELSE

        OUTPUT "We don't have that type of cake, sorry"
```

C# SWITCH /CASE STATEMENT

```
String var = "";
var = Console.ReadLine();

switch (Condition)
{
    case "1":
        code block ;
        break;
    case "2":
        code block ;
        break;
    case "3":
        code block ;
        break;

    default:
        code block ;
        break;
}
```

Condition in ( ) brackets

Code block if case is **true**

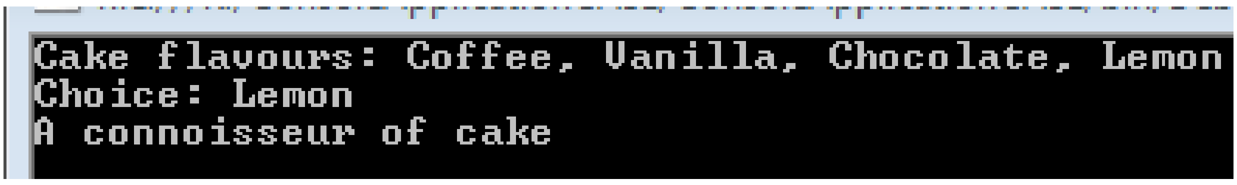
break, stops the rest of the code executing and then returns to the program

{ } brackets for the switch code cases

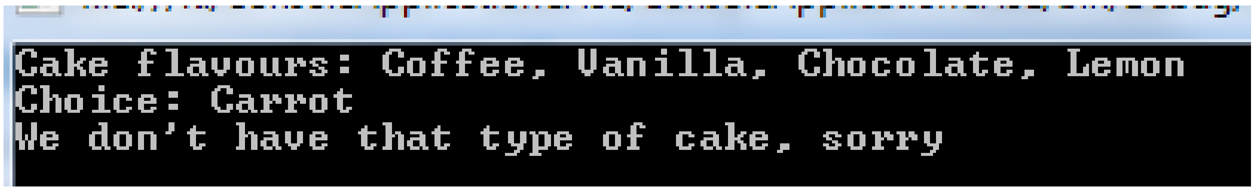
Code block if all cases are **false**

A C# CONSOLE EXAMPLE:

```
static void Main(string[] args)
{
    /* Cake input */
    String cakeFlavour = "";
    Console.WriteLine("Cake flavours: Coffee, Vanilla, Chocolate, Lemon");
    Console.Write("Choice: ");
    cakeFlavour = Console.ReadLine();

    /*Case of cakeFlavour */
    switch (cakeFlavour)
    {
        case "Coffee":
            Console.WriteLine("OK, if you must");
            break;
        case "Vanilla":
            Console.WriteLine("Excellent choice");
            break;
        case "Chocolate":
            Console.WriteLine("Very tasty");
            break;
        case "Lemon":
            Console.WriteLine("A connoisseur of cake");
            break;
        default:
            Console.WriteLine("We don’t have that type of cake, sorry");
            break;
    }
    Console.ReadLine();
}
```



```
Cake flavours: Coffee, Vanilla, Chocolate, Lemon
Choice: Lemon
A connoisseur of cake
```



```
Cake flavours: Coffee, Vanilla, Chocolate, Lemon
Choice: Carrot
We don't have that type of cake, sorry
```

NESTED IF

A nested IF is when an IF statement is placed inside another IF statement, a bit like a Russian doll.

A nested IF statement can be placed in the **true** or **false** code block and is a sophisticated branching technique, allowing a programmer to code for multiple combinations of inputs.

Working out the combination of logic can be confusing and it always worth planning how the combinations will work in advance of coding.

PSEUDOCODE

In this example the IF statement is only nested in the true **block of the first level IF.**

```
IF condition is true THEN:
    //First IF condition is true
    IF condition is true THEN:
        //First IF is true AND nested IF is true
        Do true code block
    ELSE:
        //First IF is true AND nested IF is false
        Do false code block
ELSE:
    //First IF condition is false
    Do false code block
```

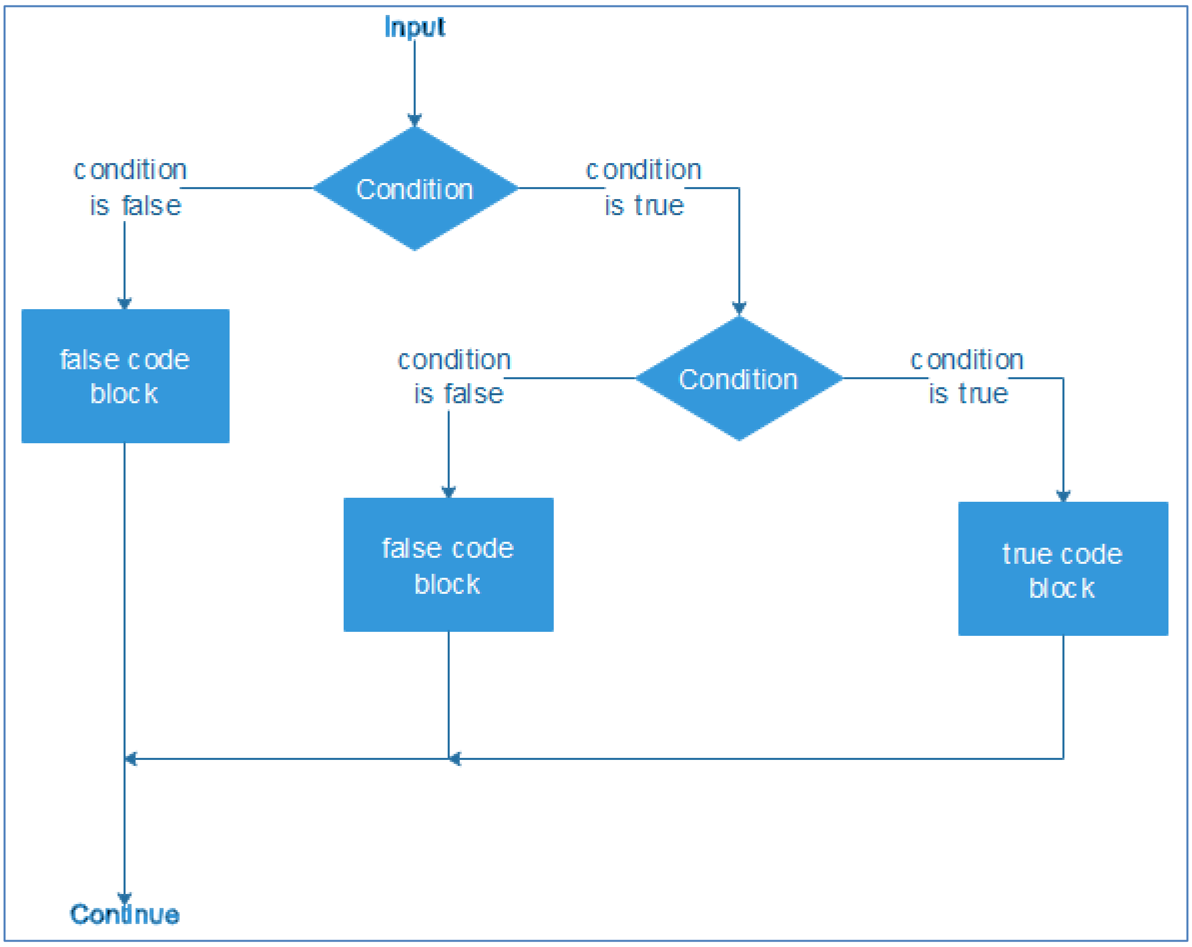
Did you notice?

The nested IF **true** block only runs if ***BOTH*** IF statements are **true**. A nested IF is dependent on the prior one being **true** first.

Note: You can nest IF statements much more than once.

FLOWCHART

Here is an example of a flowchart that describes a nested IF inside the **true** block of the first level IF.



USERNAME AND PASSWORD EXAMPLE:

An example of where a nested IF could be used is with a username and password, where the username must be correct before the password is checked. A user can only log in when BOTH conditions are found to be true.

```
username ← USERINPUT
password ← USERINPUT

IF username = “cake” THEN:

    IF password = “lemon” THEN:

        OUTPUT “Username and Password are both correct”

    ELSE:

        OUTPUT “Username is correct, password is incorrect”

ELSE:

    OUTPUT “Username is incorrect”
```

TESTING THE LOGIC

So lets test different scenarios for the example above.

USERINPUT	username = true?	password = true?	OUTPUT

username = “cheese”
password = “biscuits”
	false	Not checked	Username is incorrect

username = “cake”
password = “biscuits”
	true	false	Username is correct, password is incorrect

username = “cheese”
password = “lemon”
	false	Not checked	Username is incorrect

username = “cake”
password = “lemon”
	true	true	Username and password are both correct

As you can see, the password is only checked if the username is correct.

How could you always check the password? (answer on next page)

How could you always check the password?

Put another nested IF inside the false block for the first level IF

USERNAME AND PASSWORD EXAMPLE:

```
username ← USERINPUT
password ← USERINPUT

IF username = “cake” THEN:
    IF password = “lemon” THEN:
        OUTPUT “Username & Password are both correct”
    ELSE:
        OUTPUT “Username is correct, Password is incorrect”
ELSE:
    IF password = “lemon” THEN:
        OUTPUT “Username is incorrect & Password is correct”
    ELSE:
        OUTPUT “Username & Password are both incorrect”
```

TESTING THE LOGIC

USERINPUT	username = true?	password = true?	OUTPUT

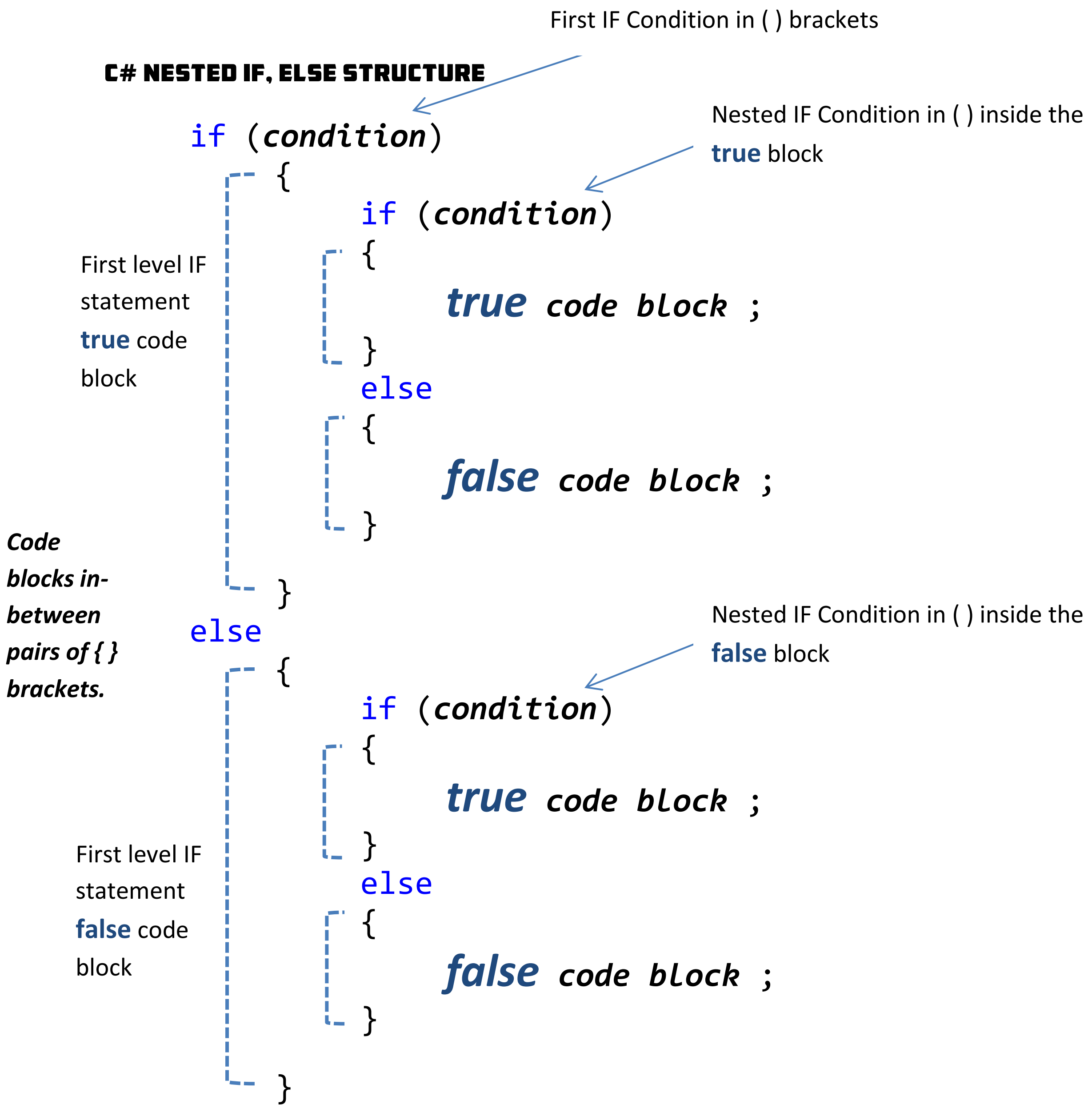
username = “cheese”
password = “biscuits”
	false	false	Username and password are both incorrect

username = “cake”
password = “biscuits”
	true	false	Username is correct, password is incorrect

username = “cheese”
password = “lemon”
	false	true	Username is correct, password is correct

username = “cake”
password = “lemon”
	true	true	Username and password are both correct

C# NESTED IF, ELSE STRUCTURE



```
if (condition)
{
    if (condition)
    {
        true code block ;
    }
    else
    {
        false code block ;
    }
}
else
{
    if (condition)
    {
        true code block ;
    }
    else
    {
        false code block ;
    }
}
```

First IF Condition in ( ) brackets
Nested IF Condition in ( ) inside the true block
First level IF statement true code block
Code blocks in-between pairs of { } brackets.
Nested IF Condition in ( ) inside the false block
First level IF statement false code block

The example above has an IF statement nested in both the **true** and **false** code blocks for the first level IF statement.

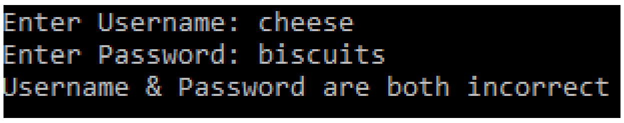
It is up to you whether you need one, both or no nesting.

A C# CONSOLE EXAMPLE:

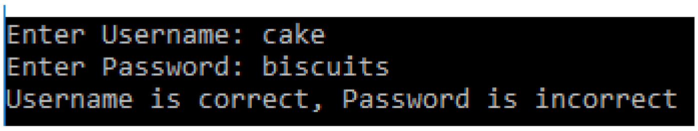
Beware of the brackets! {}

```
static void Main(string[] args)
{
    String username = "";
    String password = "";

    Console.Write("Enter Username: ");
    username = Console.ReadLine();
    Console.Write("Enter Password: ");
    password = Console.ReadLine();

    if (username == "cake")
    {
        if (password == "lemon")
        {
            Console.WriteLine("Username & Password are both correct");
        }
        else
        {
            Console.WriteLine("Username is correct, Password is incorrect");
        }
    } // end of first IF true block

    else
    {
        if (password == "lemon")
        {
            Console.WriteLine("Username is incorrect & Password is correct");
        }
        else
        {
            Console.WriteLine("Username & Password are both incorrect");
        }
    } // end of first IF false block

    //Continue program
    Console.ReadLine();
}
```

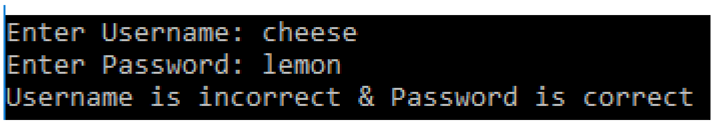
TESTING THE CODE AND LOGIC



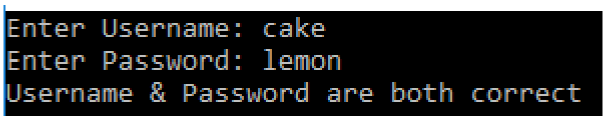
```
Enter Username: cheese
Enter Password: biscuits
Username & Password are both incorrect
```



```
Enter Username: cake
Enter Password: biscuits
Username is correct, Password is incorrect
```



```
Enter Username: cheese
Enter Password: lemon
Username is incorrect & Password is correct
```



```
Enter Username: cake
Enter Password: lemon
Username & Password are both correct
```

ACTIVITY: ANIMAL AGE

Your task is to create an animal age calculator. You will need to decide which selection method to use.

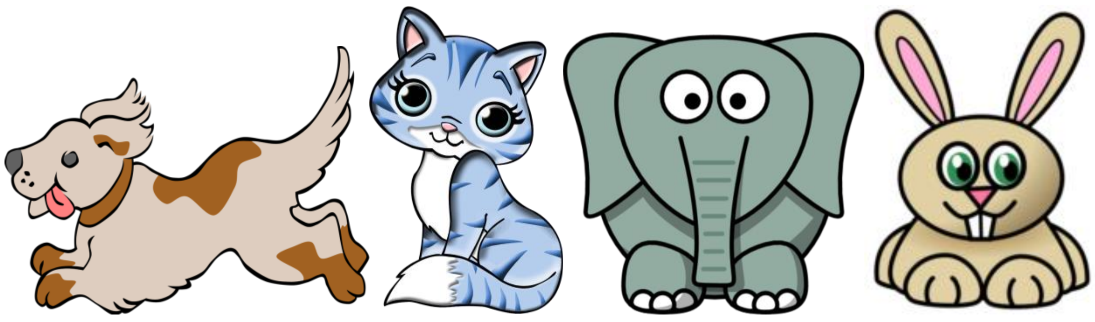
Your program will need to take in two inputs:

- Animal age required
- Human Age

The user will choose from the list which animal age that they want calculating.

One human year is approximately equivalent to:

- 3.64 dog years
- 3.2 cat years
- 1.14 elephant years
- 10 Guinea pig years
- 20 mouse years
- 8.89 rabbit years

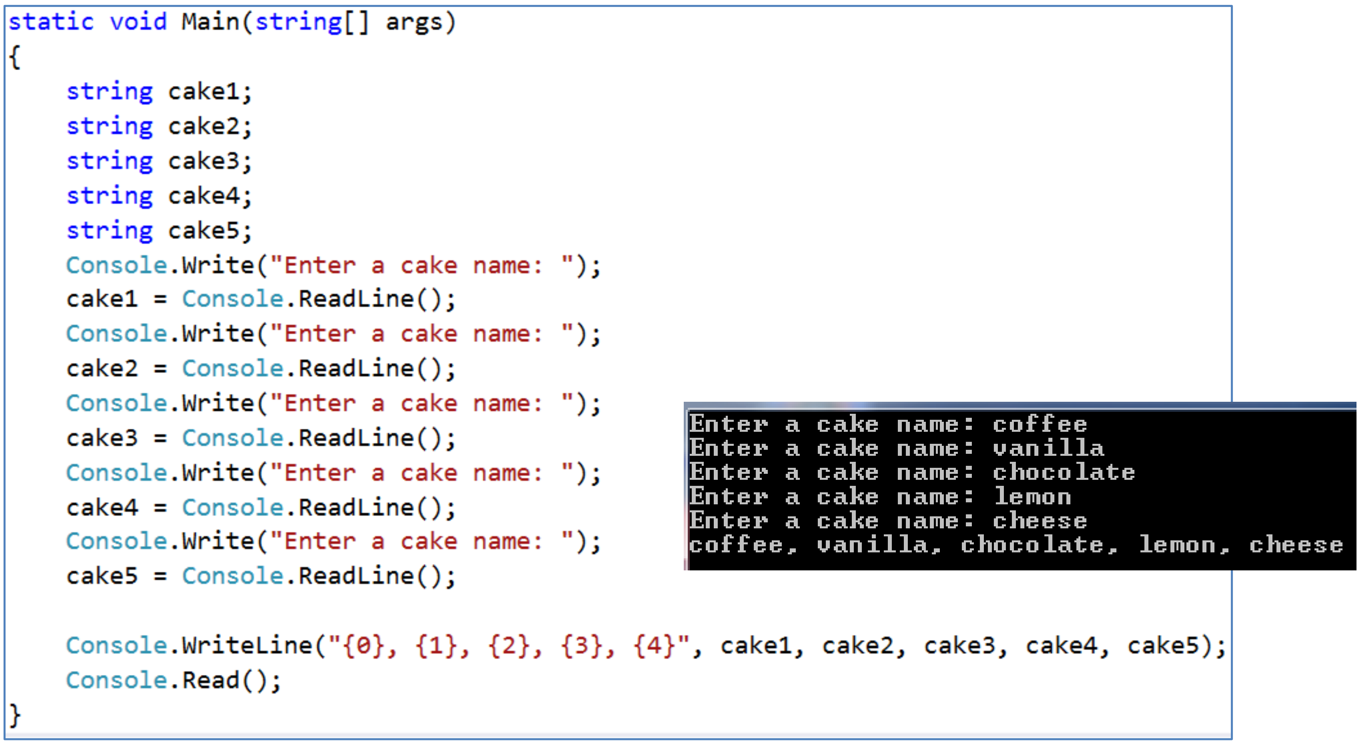


Animal age information from: https://www.easycalculation.com/other/fun/Human-years-to-Human-years.html

WHY USE AN ARRAY

Imagine you have a cake shop and you want to store a list of five cakes. You will need a variable for each cake; cake1, cake2, cake3, cake4, cake5.

Let's add the cakes.



```
static void Main(string[] args)
{
    string cake1;
    string cake2;
    string cake3;
    string cake4;
    string cake5;
    Console.Write("Enter a cake name: ");
    cake1 = Console.ReadLine();
    Console.Write("Enter a cake name: ");
    cake2 = Console.ReadLine();
    Console.Write("Enter a cake name: ");
    cake3 = Console.ReadLine();
    Console.Write("Enter a cake name: ");
    cake4 = Console.ReadLine();
    Console.Write("Enter a cake name: ");
    cake5 = Console.ReadLine();

    Console.WriteLine("{0}, {1}, {2}, {3}, {4}", cake1, cake2, cake3, cake4, cake5);
    Console.Read();
}
```

```
Enter a cake name: coffee
Enter a cake name: vanilla
Enter a cake name: chocolate
Enter a cake name: lemon
Enter a cake name: cheese
coffee, vanilla, chocolate, lemon, cheese
```

It works, but there is quite a lot of code, if you want to add another cake you will need to do quite a lot of work.

This is where arrays become useful.

Arrays are a more efficient way of storing values, can be changed more easily making your code easier to work with.

ARRAY

An array is a collection of elements that are all the same type of data. An array is a fixed size and this must be specified when it is created (initialised).

In the pseudocode example below all the elements are a **string** type and it has 5 elements:

```
cake ← ["coffee", "vanilla", "chocolate", "lemon", "cheese"]
```

Remember: Arrays need to be given a suitable name (identifier) that identifies their purpose.

INDEX

An index in a book is used to reference a word and what page it is located. This is similar in arrays; the index number is used to identify the location of an element or where to add one.

Each element in an array is has an index number starting at 0.

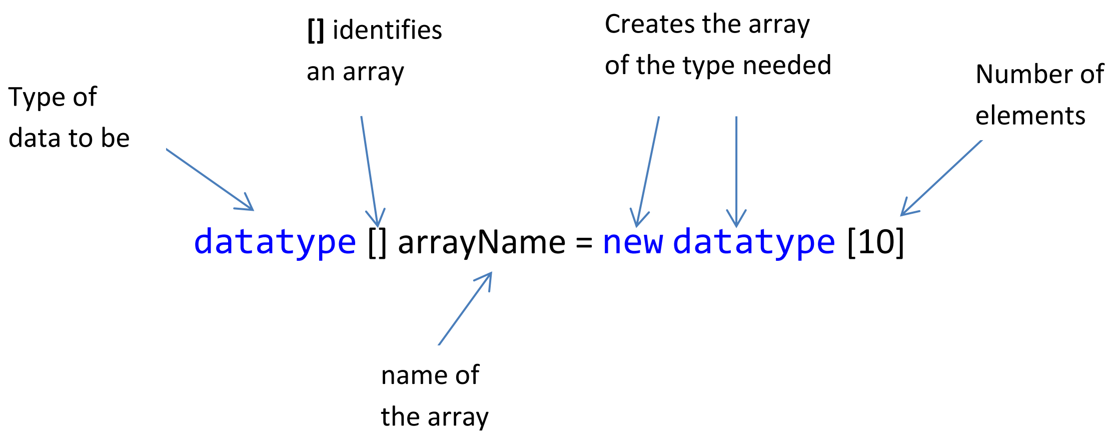
Remember 0 not 1, this is an important programming concept.

Here is an example:

index	*0*	*1*	*2*	*3*	*4*
Element	**“coffee”**	**“vanilla”**	**“chocolate”**	**“lemon”**	**“cheese”**

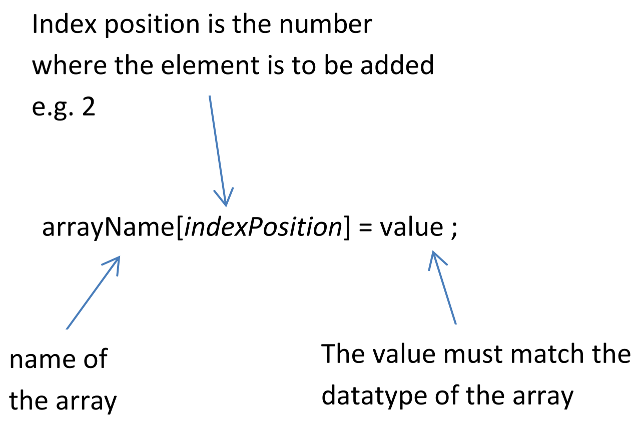
C# ARRAY STRUCTURE

Setting up an array



[] identifies an array
Creates the array of the type needed
Type of data to be
Number of elements
datatype [] arrayName = new datatype [10]
name of the array

Adding elements to an array

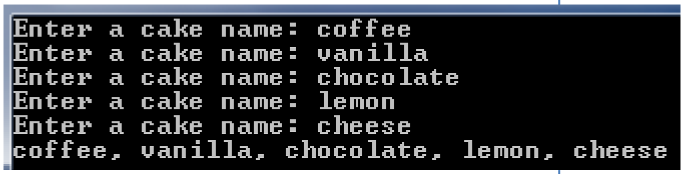


Index position is the number where the element is to be added e.g. 2
arrayName[*indexPosition*] = value ;
name of the array
The value must match the datatype of the array

C# EXAMPLE

In this example the size of the array is fixed at 5.

```
static void Main(string[] args)
{
    string[] cake = new string[5];
    Console.Write("Enter a cake name: ");
    cake[0] = Console.ReadLine();
    Console.Write("Enter a cake name: ");
    cake[1] = Console.ReadLine();
    Console.Write("Enter a cake name: ");
    cake[2] = Console.ReadLine();
    Console.Write("Enter a cake name: ");
    cake[3] = Console.ReadLine();
    Console.Write("Enter a cake name: ");
    cake[4] = Console.ReadLine();

    Console.WriteLine("{0}, {1}, {2}, {3}, {4}", cake[0], cake[1], cake[2], cake[3], cake[4]);
    Console.Read();
}
```



```
Enter a cake name: coffee
Enter a cake name: vanilla
Enter a cake name: chocolate
Enter a cake name: lemon
Enter a cake name: cheese
coffee, vanilla, chocolate, lemon, cheese
```

Limitations

- There is repetitive code and we programmers hate repeating code.
- The size of the array is fixed, what if the user wants a different number of cakes.

There is a better way but first you must learn about iteration. Look in the FOR loop section to see how to improve this code.

ITERATION

The third key programming technique is **iteration** or repetition. This is often seen as the most complex technique because you need to work out the logic required.

You may have heard the programming acronym DRY (Don't Repeat Yourself), we programmers ideally don't like to type the same code in more than once. It is quite common that the same code needs to run again and again, this is called iteration or looping. So using iteration techniques mean we don't have to type in the same code again and again and again and again, you get the idea.

There are three main types of iteration; you will need to decide when to use each type:

- FOR ….. a number of times do something

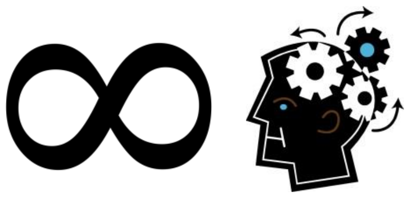
```
FOR a number of times
    Do something
END FOR
```

- WHILE …..a condition is true do something

```
WHILE condition is true
    Do something
END WHILE
```

- REPEAT …….. until a condition is true do something

```
REPEAT
    Do something
UNTIL condition is true
```



INFINITE LOGIC PROBLEMS

Just before you get into the different types of loop, a word of warning!

The WHILE and REPEAT loops both use a condition, if you get your logic wrong you can create an infinite loop, which is one that never ends. In fact, it will continue until the computer runs out of memory or you stop the program.

INFINITE EXAMPLE

condition: IF the variable *number* is less than 10

true

```
number ←1
WHILE number <= 10 THEN:
    OUTPUT "I'm going loopy"
    number = number -1
#Continue here when the condition is false
OUTPUT "I've stopped"
```

Run the code

LOGIC ERROR

The condition **number <= 10** will never be **false**.

Why?

number = number -1 is counting down each loop, -2, -3, -4 ….. and will never be greater than 10.

This means that the condition **number <= 10** will always be **true** and will run the code forever creating an infinite loop.

FOR LOOP

A for loop needs to know how many times to loop *before* it starts e.g. the length of a string or array, a user input.

There is no condition, the loop will repeat a specific number of times and is controlled by a loop counter variable that is compared to maximum number of iterations needed.

The loop counter can increase in value (increment ***counter++***) or decrease in value (decrement **--*counter***).

WHEN TO USE

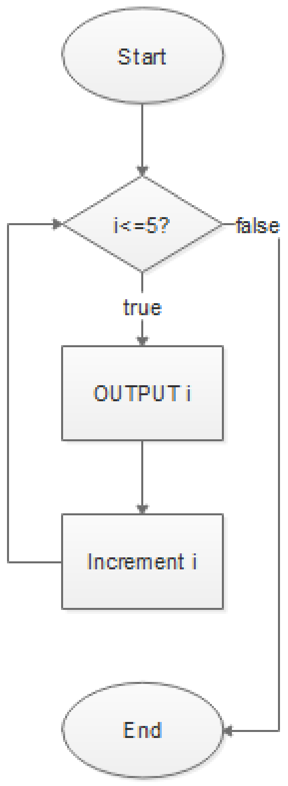
- If you ***know how many times*** you want the code to repeat in total

EXAMPLES

- Calculate the first 10 numbers in the times table for a number given by a user
- Find the length of an array, string of text or file and loop through to perform an action

PSEUDOCODE AND FLOWCHART

```
FOR i ← 1 TO 5
    OUTPUT i
ENDFOR

# this will output: 1, 2, 3, 4 and 5
```



C# FOR LOOP STRUCTUR

Initialise (setup) the loop counter variable

The condition checks the counter variable and if true runs the code statements, otherwise skips the code block and returns to the main program

Increment increases the loop counter variable, often using ++ which adds 1

```
for (initialise; condition; increment)
     {
             code statements ;
     }
```

Code statements to repeat are placed between { } brackets

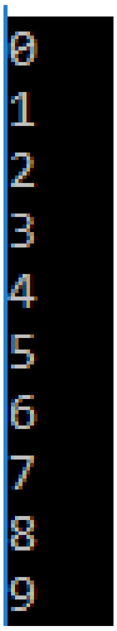
PSEUDOCODE EXAMPLE:

```
FOR counter ← 0 TO 9
    OUTPUT counter
ENDFOR
```

A C# CONSOLE EXAMPLE:

THE FOLLOWING PROGRAM LOOPS 10 TIMES (0 TO 9) AND OUTPUTS THE COUNTER VALUE.

```
static void Main(string[] args)
{
    for (int counter = 0; counter < 10; counter++)
    {
        Console.WriteLine(counter);
    }

    Console.ReadLine();
}
```



```
0
1
2
3
4
5
6
7
8
9
```

Program OUTPUT

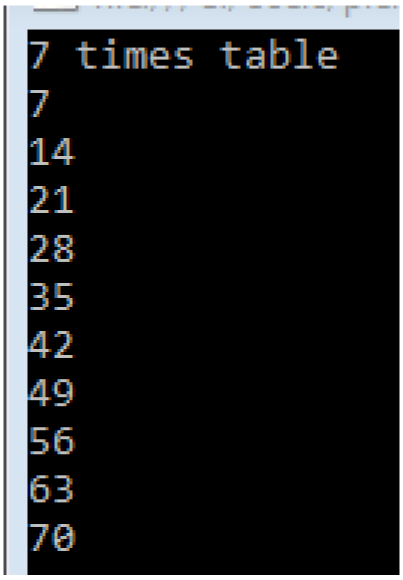
ACTIVITY: TIMES TABLE

Create a console program to do the 10 times table for a number given by a user

- Ask the user to input a number between 1 and 10
- Output the times table by multiplying the user number by the counter number
- You will need to change the counter start and end values to make it work

Note: Don't forget to use parse to cast the user input from a string to an integer

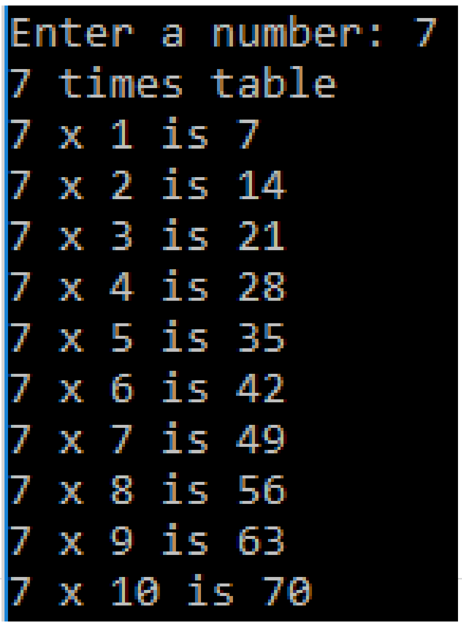
Example output:



```
7 times table
7
14
21
28
35
42
49
56
63
70
```

TIMES TABLE EXTENSION ACTIVITY:

Format the output to be the same as below but with the ability for the user to choose the number.



```
Enter a number: 7
7 times table
7 x 1 is 7
7 x 2 is 14
7 x 3 is 21
7 x 4 is 28
7 x 5 is 35
7 x 6 is 42
7 x 7 is 49
7 x 8 is 56
7 x 9 is 63
7 x 10 is 70
```

Programming techniques required:

- Variables
- Casting using parse for user input
- String formatting using placeholders

A C# CONSOLE EXAMPLE, WITH AN ARRAY USING THE INDEX:

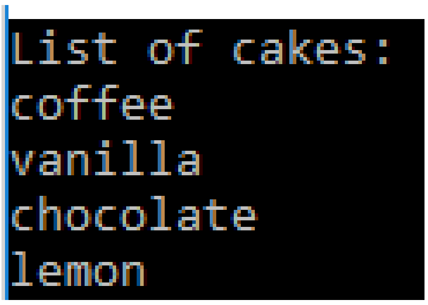
The following program loops through the cake array, from index position 0 to the length of the array and outputs the string value for each index position.

PSEUDOCODE:

```
        cake ← ["coffee", "vanilla", "chocolate", "lemon", "cheese"]

        FOR counter ← 0 TO LEN(cake)

                OUTPUT cake[counter]

        ENDFOR
```

C# CONSOLE CODE:

```
static void Main(string[] args)
{
    String [] cake = {"coffee", "vanilla", "chocolate", "lemon" };

    Console.WriteLine("List of cakes: ");

    for (int counter = 0; counter < cake.Length; counter++)
    {
        Console.WriteLine(cake[counter]);
    }

    Console.ReadLine();
}
```



```
List of cakes:
coffee
vanilla
chocolate
lemon
```

Program OUTPUT

A SIMPLER WAY

C# has a simpler way of looping through items in an array, the FOR EACH loop.

FOREACH LOOP

The FOREACH loop is an easy way to get each element from an array.

- FOREACH ….. item in the array

```
FOR each item IN myArray
    Do something
END FOR
```

C# FOREACH LOOP STRUCTURE

The *item* variable will temporarily store each item in the array

The datatype must match the type of data in the array; **int**, **string**, **char** etc.

Name of the array

```
foreach (datatype item in array)
        code statements ;
```

Code statements are NOT between { } brackets. They must close with a ;

As you can see it is much simpler to setup.

LIMITATIONS

A FOREACH also has its limitations:

- There is no counter variable that can be used
- You cannot control where to start, it always starts at the first item
- You cannot loop through an array in reverse, you can with a standard FOR

A C# CONSOLE EXAMPLE, WITH AN ARRAY USING THE INDEX:

Here is the same example used with a standard FOR loop but using FOREACH instead.

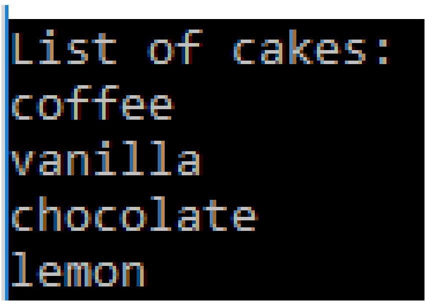
PSEUDOCODE:

```
cake ← [“coffee”, “vanilla”, “chocolate”, “lemon”]
FOR item IN cake
    OUTPUT item
ENDFOR
```

C# CONSOLE CODE:

```
static void Main(string[] args)
{
    string[] cake = { "coffee", "vanilla", "chocolate", "lemon" };

    Console.WriteLine("List of cakes: ");

    foreach (string item in cake)
        Console.WriteLine(item);

    Console.ReadLine();
}
```

As you can see, the same result with less code.



```
List of cakes:
coffee
vanilla
chocolate
lemon
```

Program OUTPUT

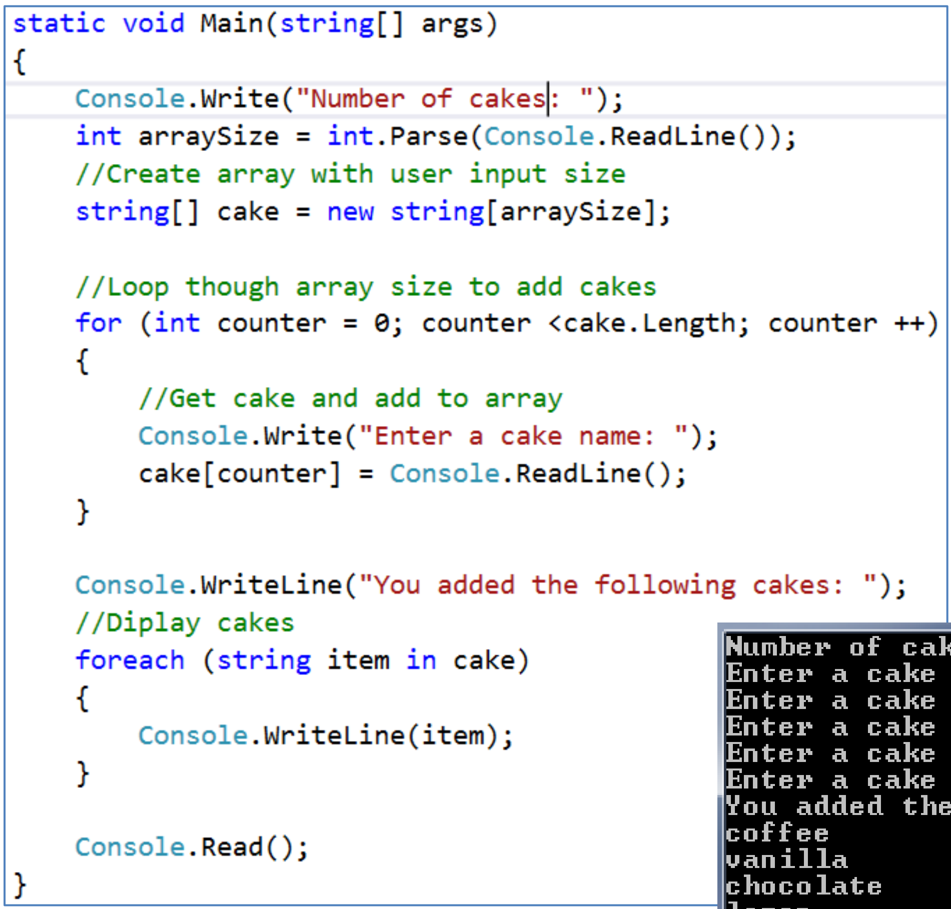
A C# CONSOLE EXAMPLE, ADDING TO AND DISPLAYING AN ARRAY USING THE INDEX:

Here is an example of adding items to an array using a FOR loop and displaying the result

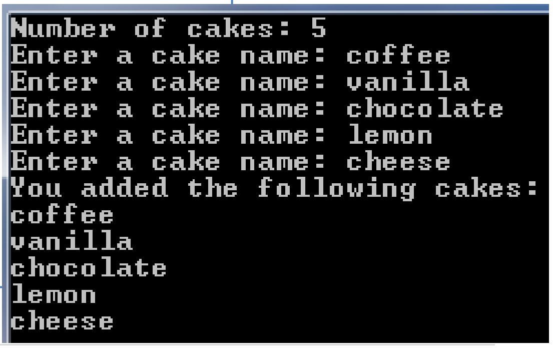
PSEUDOCODE:

```
cake ← []
OUPUT “Number of cakes: “
arraySize ← USERINPUT

FOR counter ← 0 TO  arraySize
        OUTPUT “Enter a cake: “
        cake[counter] = USERINPUT
ENDFOR


OUPUT “You entered the following cakes:  “
FOR item IN cake
        OUTPUT item
ENDFOR
```

C# CONSOLE CODE:



```
static void Main(string[] args)
{
    Console.Write("Number of cakes: ");
    int arraySize = int.Parse(Console.ReadLine());
    //Create array with user input size
    string[] cake = new string[arraySize];

    //Loop though array size to add cakes
    for (int counter = 0; counter <cake.Length; counter ++)
    {
        //Get cake and add to array
        Console.Write("Enter a cake name: ");
        cake[counter] = Console.ReadLine();
    }

    Console.WriteLine("You added the following cakes: ");
    //Diplay cakes
    foreach (string item in cake)
    {
        Console.WriteLine(item);
    }

    Console.Read();
}
```



```
Number of cakes: 5
Enter a cake name: coffee
Enter a cake name: vanilla
Enter a cake name: chocolate
Enter a cake name: lemon
Enter a cake name: cheese
You added the following cakes:
coffee
vanilla
chocolate
lemon
cheese
```

LOGICAL OPERATORS

When your programming becomes more advanced you will probably need to work with logical operators. Here is a quick introduction of AND, OR and NOT.

&& AND

A logical AND returns true when both conditions are true. The operator used in C# is a &&.

Input	Logical	C#	Result	Why

a = true
b = true
	a **AND** b	a **&&** b	true	Both a and b are true

a = true
b = false
	a **AND** b	a **&&** b	false	b is false, so both are not true

a = 1
b = 10
	a >0 **AND** b <11	a **&&** b	true	a = 1, 1>0 = true
				b = 10, 10<11 = true
				
				So both conditions are true

a = 1
b = 12
	a >0 **OR** b <11	a **&&** b	false	a = 1, 1>0 = true
				b = 10, 10<12 = false
				
				So both conditions are not true

|| OR

A logical OR returns true if either of the conditions are true. The operator used in C# is a ||.

Input	Logical	C#	Result	Why

a = true
b = true
	a **OR** b	a **||** b	true	Both a and b are true

a = true
b = false
	a **OR** b	a **||** b	true	b is false, but a is true so at least one condition is true

a = 1
b = 10
	a >0 **OR** b <11	a **||** b	true	a = 1, 1>0 = true
				b = 10, 10<11 = true
				
				So both conditions are true

a = 0
b = 12
	a >0 **OR** b <11	a **||** b	false	a = 0, 0>0 = false
				b = 10, 10<12 = false
				
				So neither conditions are true

! NOT

A true condition becomes false and vice versa. The operator used in C# is a !.

Input	Logical	C#	Result	Why
a = true	**NOT** a	**!**a	false	a is true and becomes opposite
a = false	**NOT** a	**!**a	true	a is false and becomes opposite

a = true
b = false
	a **AND** !b	a **&&** **!**b	true	a is true.
				b is false, but becomes true.
				
				So both conditions are true

WHILE LOOP

A while loop does not need to know how many times to repeat before it starts, it will decide using a condition if to run at all and will keep checking this condition ***before*** each loop.

You can think of a while loop having an IF statement that is checked at the beginning of a loop before running the code.

There is a condition at the start of the loop. If the condition isn't met at the start the loop will not execute.

WHEN TO USE

- If you ***DON'T know how many times*** you want the code to repeat total
- You ***only*** want to run the code ***if a condition*** is met

EXAMPLES

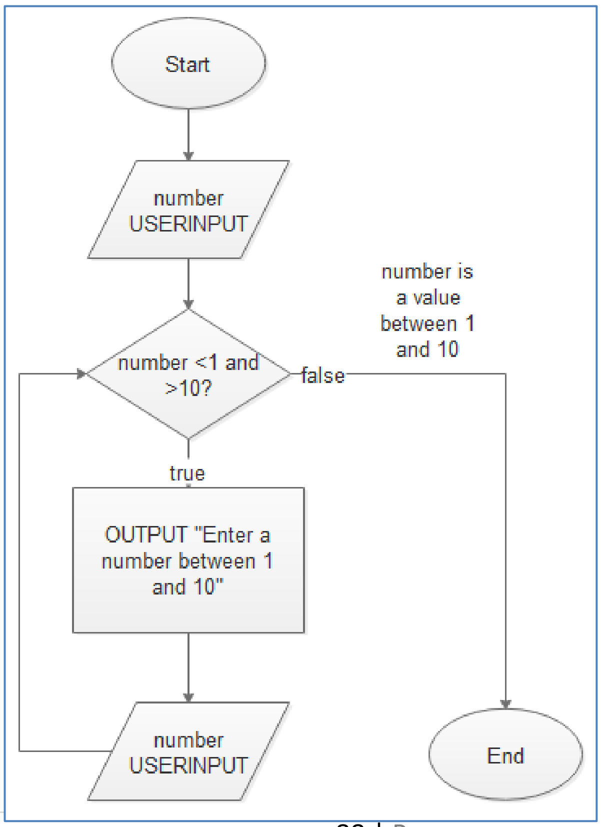
THERE ARE MANY SITUATIONS WHERE A WHILE LOOP CAN BE USED

- When you want to keep asking a user for a value until they choose one that is valid
- To add an unknown number of items to an array, with the user deciding when to stop
- Create a menu of options and only stops when a user chooses to quit.

PSEUDOCODE AND FLOWCHART

The following is an example of validation and will keep asking a user for a number between 1 and 10.

```
number ← USERINPUT
WHILE number <1 OR number >10
OUTPUT "Enter a number between 1 and 10"
    number ← USERINPUT
ENDWHILE
```


Start
number USERINPUT
number <1 and >10?
false
number is a value between 1 and 10
true
OUTPUT "Enter a number between 1 and 10"
number USERINPUT
End

Code statements only execute if the condition is **true**

C# WHILE LOOP STRUCTURE

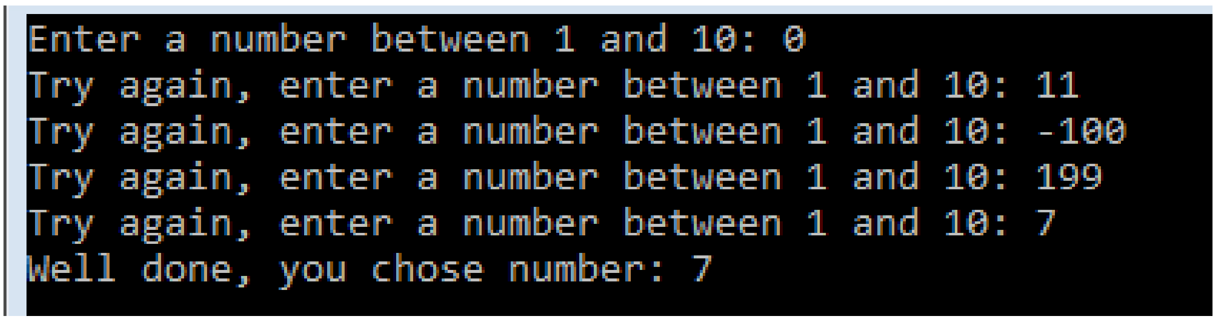
```
while (condition)
    {
        code statements ;


    }
```

Code statements to repeat are placed between { } brackets

A C# CONSOLE EXAMPLE:

The following code will keep asking a user for number between 1 and 10.

```
static void Main(string[] args)
{
    int number = 0;

    Console.Write("Enter a number between 1 and 10: ");
    number = int.Parse(Console.ReadLine());

    //Condition checks newCake variable is not empty
    while (number < 1 || number > 10)
    {
        Console.Write("Try again, enter a number between 1 and 10: ");
        number = int.Parse(Console.ReadLine());
    }

    Console.WriteLine("Well done, you chose number: {0}", number);
    Console.ReadLine();
}
```



```
Enter a number between 1 and 10: 0
Try again, enter a number between 1 and 10: 11
Try again, enter a number between 1 and 10: -100
Try again, enter a number between 1 and 10: 199
Try again, enter a number between 1 and 10: 7
Well done, you chose number: 7
```

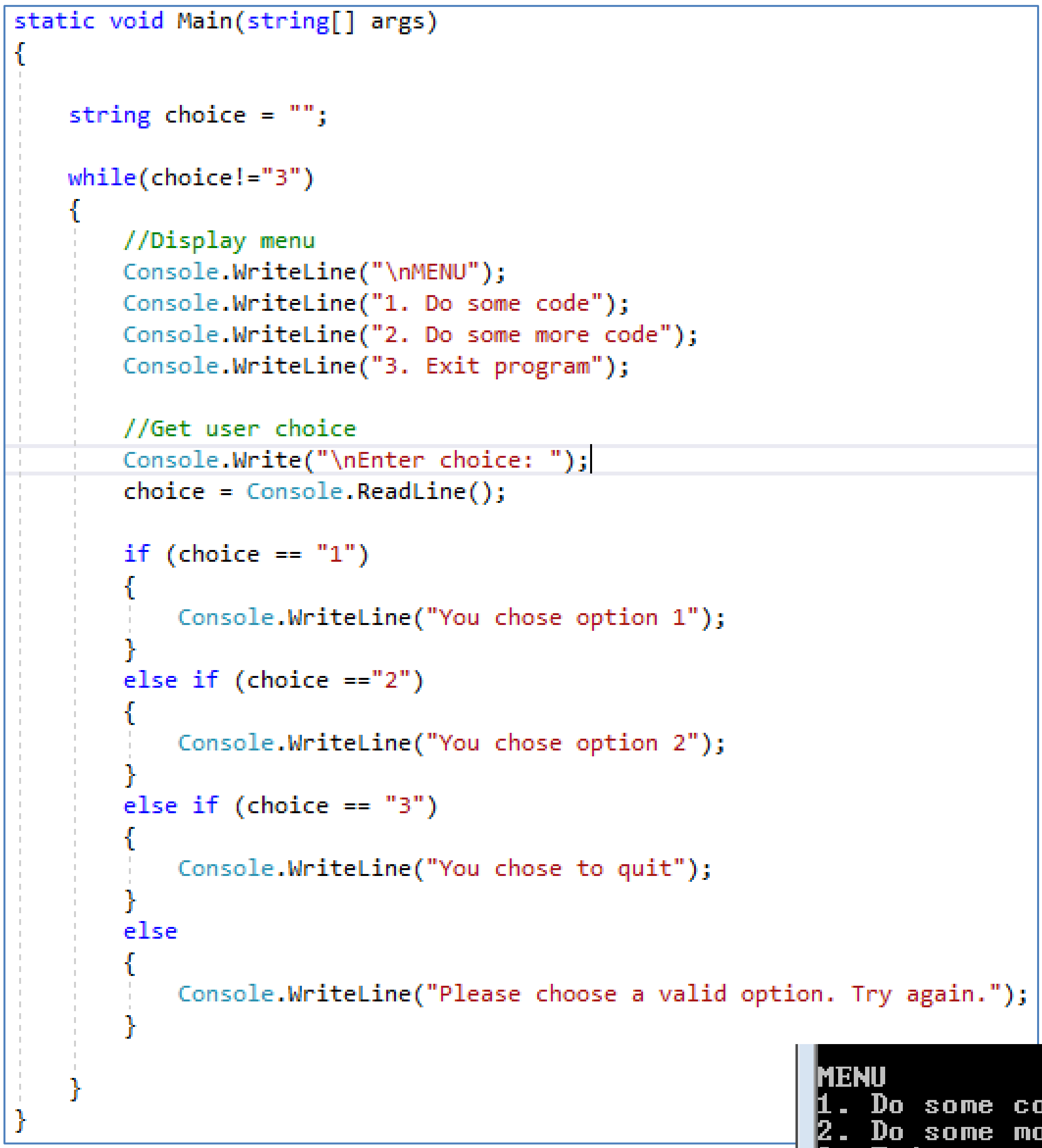
PSEUDOCODE – MENU EXAMPLE

The following is an example of a menu and will keep repeating until a user enters 3 to quit. The IF statement would contain the code required for each option, in this example it just outputs a message.

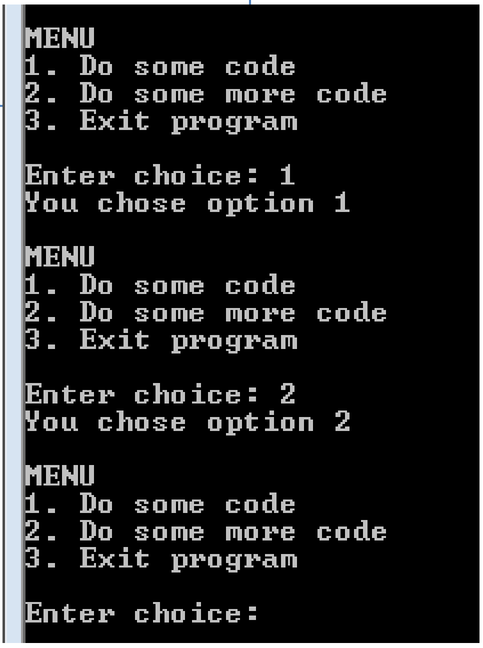
```
choice ← 0

WHILE number !=3 DO

    OUTPUT "MENU"
    OUTPUT "1. Do some code"
    OUTPUT "2. Do some other code"
    OUTPUT "3. Exit program"


    OUTPUT "Enter choice: "
    choice ← USERINPUT


    IF (choice = "1") THEN

        OUTPUT "You chose option 1"

    ELSEIF (choice = "2")

        OUTPUT "You chose option 2"

    ELSEIF (choice = "3")

        OUTPUT "You chose to quit"

    ELSE

        OUTPUT "Please choose a valid option. Try again"

    ENDIF

ENDWHILE
```

MENU C# CONSOLE EXAMPLE:

The following code will keep asking a user for number between 1 and 10.



```
static void Main(string[] args)
{

    string choice = "";

    while(choice!="3")
    {
        //Display menu
        Console.WriteLine("\nMENU");
        Console.WriteLine("1. Do some code");
        Console.WriteLine("2. Do some more code");
        Console.WriteLine("3. Exit program");

        //Get user choice
        Console.Write("\nEnter choice: ");
        choice = Console.ReadLine();

        if (choice == "1")
        {
            Console.WriteLine("You chose option 1");
        }
        else if (choice =="2")
        {
            Console.WriteLine("You chose option 2");
        }
        else if (choice == "3")
        {
            Console.WriteLine("You chose to quit");
        }
        else
        {
            Console.WriteLine("Please choose a valid option. Try again.");
        }

    }
}
```



```
MENU
1. Do some code
2. Do some more code
3. Exit program

Enter choice: 1
You chose option 1

MENU
1. Do some code
2. Do some more code
3. Exit program

Enter choice: 2
You chose option 2

MENU
1. Do some code
2. Do some more code
3. Exit program

Enter choice:
```

REPEAT (DO …WHILE) LOOP

A repeat loop does not need to know how many times to loop before it starts but will always loop once. It will decide using a condition if to stop after the first loop and will keep checking this condition ***after*** each loop before looping again.

There is a condition at the end of the loop. The loop will always execute at least once.

Note: There is no repeat loop in C# it is called a do...while

WHEN TO USE

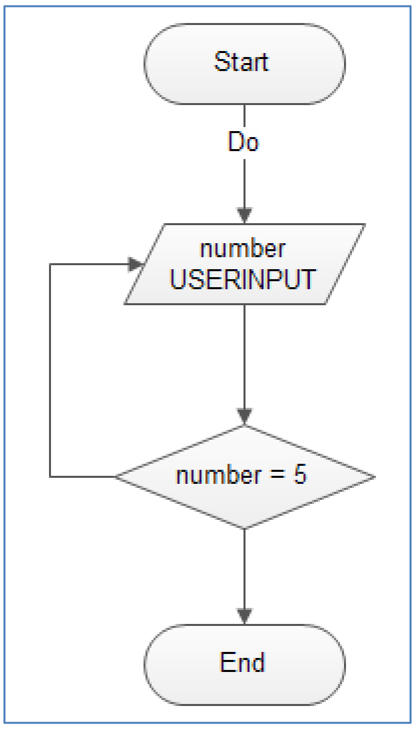
- If you ***DON'T know how many times*** you want the code to repeat in total
- But you ***always*** want to run the code ***at least once*** if a condition is met

EXAMPLES

- Display a menu of choices once, keep displaying until close program option is chosen
- Until a correct/valid value within a range is entered

PSEUDOCODE

```
REPEAT
    OUTPUT "Guess an number between 1 and 5: "
    num ← USERINPUT
    OUTPUT num
UNTIL num = 5
```



C# DO WHILE LOOP STRUCTURE

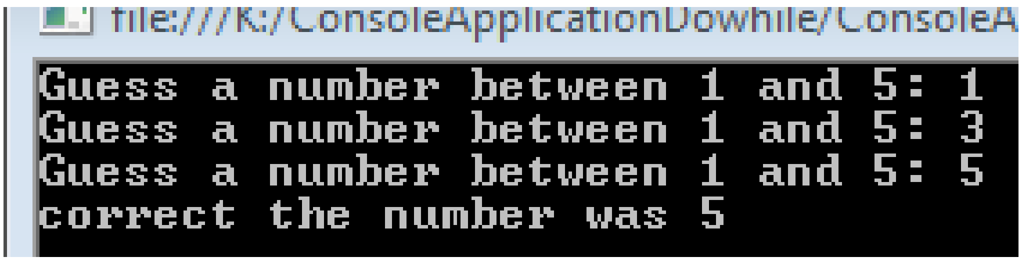
```
do
      {
            code statements ;

      } while (condition);
```

Code statements to repeat are placed between { } brackets

Code statements only execute UNTIL if the condition is **true**

A C# CONSOLE EXAMPLE:

```
static void Main(string[] args)
{
    int num;

    do
    {
        Console.Write("Guess a number between 1 and 5: ");
        num = int.Parse(Console.ReadLine());

    } while (num != 5);

    Console.WriteLine("correct the number was 5");
    Console.Read();
}
```



```
file:///K:/ConsoleApplicationDowhile/ConsoleA
Guess a number between 1 and 5: 1
Guess a number between 1 and 5: 3
Guess a number between 1 and 5: 5
correct the number was 5
```

ADVANCED STRINGS AND LOOPS

When working with strings, sometimes there is a need to loop through each character to perform a check or action on it. A typical example would be to see if a character is allowed, like in a username or password.

OPTION 1 - SELECTION

Create an IF, ELSE IF selection structure for each of the characters allowed / not allowed and check each character in a string using a loop.

OPTION 2 - ARRAY

Create an array data structure that contains each of the characters allowed / not allowed and check each character in a string using a loop.

OPTION 3 – NUMERICAL ASCII VALUE

Convert each character to a numeric ASCII value and compare with the range of numbers allowed.

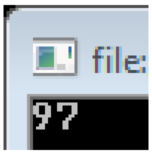
ASCII Table

Dec	Char	Description	Dec	Char	Description	Dec	Char	Description
32		space	64	@	At sign	96	`	Grave accent
33	!	exclamation mark	65	A	Capital A	97	a	Lowercase a
34	"	Quotation mark	66	B	Capital B	98	b	Lowercase b
35	#	Number sign	67	C	Capital C	99	c	Lowercase c
36	$	Dollar sign	68	D	Capital D	100	d	Lowercase d
37	%	Percent sign	69	E	Capital E	101	e	Lowercase e
38	&	Ampersand	70	F	Capital F	102	f	Lowercase f
39	'	Apostrophe	71	G	Capital G	103	g	Lowercase g
40	(	round brackets or parentheses	72	H	Capital H	104	h	Lowercase h
41	)	round brackets or parentheses	73	I	Capital I	105	i	Lowercase i
42	*	Asterisk	74	J	Capital J	106	j	Lowercase j
43	+	Plus sign	75	K	Capital K	107	k	Lowercase k
44	,	Comma	76	L	Capital L	108	l	Lowercase l
45	-	Hyphen	77	M	Capital M	109	m	Lowercase m
46	.	Full stop , dot	78	N	Capital N	110	n	Lowercase n
47	/	Slash	79	O	Capital O	111	o	Lowercase o
48	0	number zero	80	P	Capital P	112	p	Lowercase p
49	1	number one	81	Q	Capital Q	113	q	Lowercase q
50	2	number two	82	R	Capital R	114	r	Lowercase r
51	3	number three	83	S	Capital S	115	s	Lowercase s
52	4	number four	84	T	Capital T	116	t	Lowercase t
53	5	number five	85	U	Capital U	117	u	Lowercase u
54	6	number six	86	V	Capital V	118	v	Lowercase v
55	7	number seven	87	W	Capital W	119	w	Lowercase w
56	8	number eight	88	X	Capital X	120	x	Lowercase x
57	9	number nine	89	Y	Capital Y	121	y	Lowercase y
58	:	Colon	90	Z	Capital Z	122	z	Lowercase z
59	;	Semicolon	91	[	square brackets or box brackets	123	{	curly brackets or braces
60	<	Less-than sign	92	\	Backslash	124	|	vertical-bar, vbar, vertical line or vertical slash
61	=	Equals sign	93	]	square brackets or box brackets	125	}	curly brackets or braces
62	>	Greater-than sign ; Inequality	94	^	Caret or circumflex accent	126	~	Tilde ; swung dash
63	?	Question mark	95	_	underscore, under strike, underbar or low line			

CHARACTER ASCII VALUE

A variable setup as an **int** datatype will automatically convert a character into an integer number.

```
int num;
num = 'a';
Console.WriteLine(num);
```

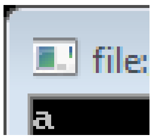


```
97
```

ASCII VALUE TO CHARACTER

A variable setup as a **char** datatype will automatically convert an integer to a character.

```
char character;
character = Convert.ToChar(97);
Console.WriteLine(character);
```



```
a
```

C# Example – Username Check

A school username is your first initial, surname and year you joined, for example, Joe Bloggs joining in 2015 is *jbloggs15*.

The following program loops through a username and checks to see if it only has lowercase letters from the alphabet and numbers, other symbols are not valid.

It converts each character to the ASCII value and checks if the number is in the correct range of characters.

Pseudocode

The following is an example of the pseudocode.

```
username ← USERINPUT

FOR EACH char IN username

    value ← GetAsciiNum(char)

    IF (value >=48 AND value <=57) OR (value >=98 AND value <=122) THEN

        OUTPUT “char” + “is a Valid character”

    ELSE

        OUTPUT “char” +  “is an invalid character”

    ENDIF

ENDFOREACH
```

A C# CONSOLE EXAMPLE:

```
Console.WriteLine(num);
Console.WriteLine();
string username;
int asciiNum;

//get username input
Console.Write("Enter username: ");
username = Console.ReadLine();

//loop through each username character
foreach (char letter in username)
{
    //letter is converted into ASCII value
    asciiNum = letter;

    //Check to see if ascii value is in the allowed range
    //between 48 and 57 OR between 97 and 122
    if ((asciiNum >=48 && asciiNum <=57) || (asciiNum >= 98 && asciiNum <= 122))
    {
        Console.WriteLine("{0} is a valid character", letter);
    }
    else
    {
        Console.WriteLine("{0} is an invalid character", letter);

    }//end of if

}//end of foreach

Console.Read();
```

Note: here is an alternative way to check a range of values using Enumerable

```
int num = 122;
if (Enumerable.Range(97, 122).Contains(num))
{
    Console.WriteLine("{0} is a valid character", num);
}
else
{
    Console.WriteLine("{0} is an invalid character", num);
}
```

An enumerable is a set (or range) of positive integers that can be counted one, by one.

SUBROUTINES - CREATING A PROGRAM STRUCTURE

So far you have been creating programs that are a loosely organised set of code and have tried to group sections together. Think of this as a routine, a set of steps, like of a set of steps in a dance.

SUBROUTINES

The way the code has been written so far means, that if you want to do the same task again you would write the code twice. Hang on a minute, we programmers a lazy, the less typing the better, remember DRY (Don't Repeat Yourself) from earlier.

This is where **subroutines** come in. You can give a block of code a name and use it again and again and again, without rewriting the code, nice!

A bit like a smaller set of steps in a dance that are repeated, again and again.

FUNCTIONS VS PROCEDURES VS METHODS

So far I have been talking about **subroutines**, you may have heard about **functions**, **procedures** and **methods**. What is the difference?

Well they are ways of creating **subroutines**. You know us programmers like to sound clever!

PROCEDURE

A **procedure** is a type of subroutine that performs a task but ***does not*** have to return a value to the main program. For example:

- Output a list of options or a menu
- Output the list of an array

FUNCTION

A **function** is a type of subroutine that performs a task that ***always*** returns a value to the main program. It is common for a function to perform a calculation. For example:

- Calculate a total from two inputs

METHOD

A **method** is an alternative way of saying subroutine and can be either a **procedure** or **function**. The C# programming language refers to subroutines as **methods**.

Methods are part of a style of programming called Object-Oriented; If you continue your programming journey you will no doubt learn all about this technique.

NAMING METHODS

The following rules are recommended for giving methods an **identifier** (name):

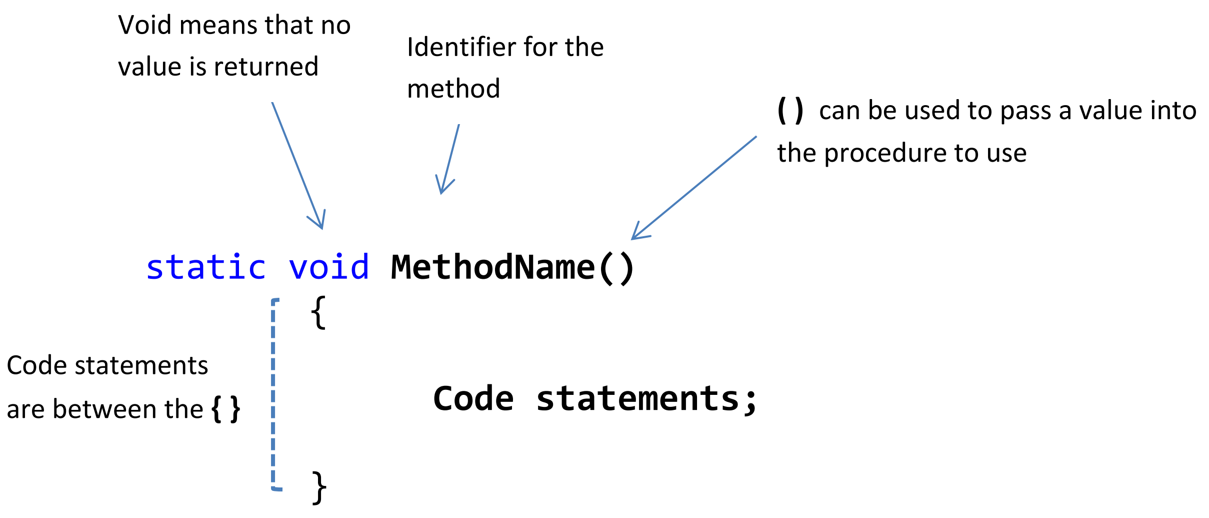
- Use verbs or verb phrases to name methods.
  - ListItems
  - GetValue
  - CalculateTotal
- Use PascalCase
  - Each word used is Capitalized

C# METHOD STRUCTURE

C# refers to subroutines as methods and there are two basic types

PROCEDURE

A **procedure** is a type of subroutine that performs a task but ***does not*** have to return a value to the main program.


Void means that no value is returned
Identifier for the method
( ) can be used to pass a value into the procedure to use
Code statements are between the { }

```
static void MethodName()
    {

        Code statements;

    }
```

Procedure example:

Identifier for the method

```
static void OutputHello()
        {
         Console.WriteLine("Hello");

        }
```

Procedure example with a value:

int indicates the type of value required

Identifies the **parameter** name available for the code statements

```
static void OutputAge(int number)
        {
         Console.WriteLine("You input {0}", number);

        }
```

number is used here

PROCEDURE CONFUSION!

Hang on!

A **procedure** is a type of subroutine that performs a task but ***does not*** have to return a value. Surely the example ***returns the number***!

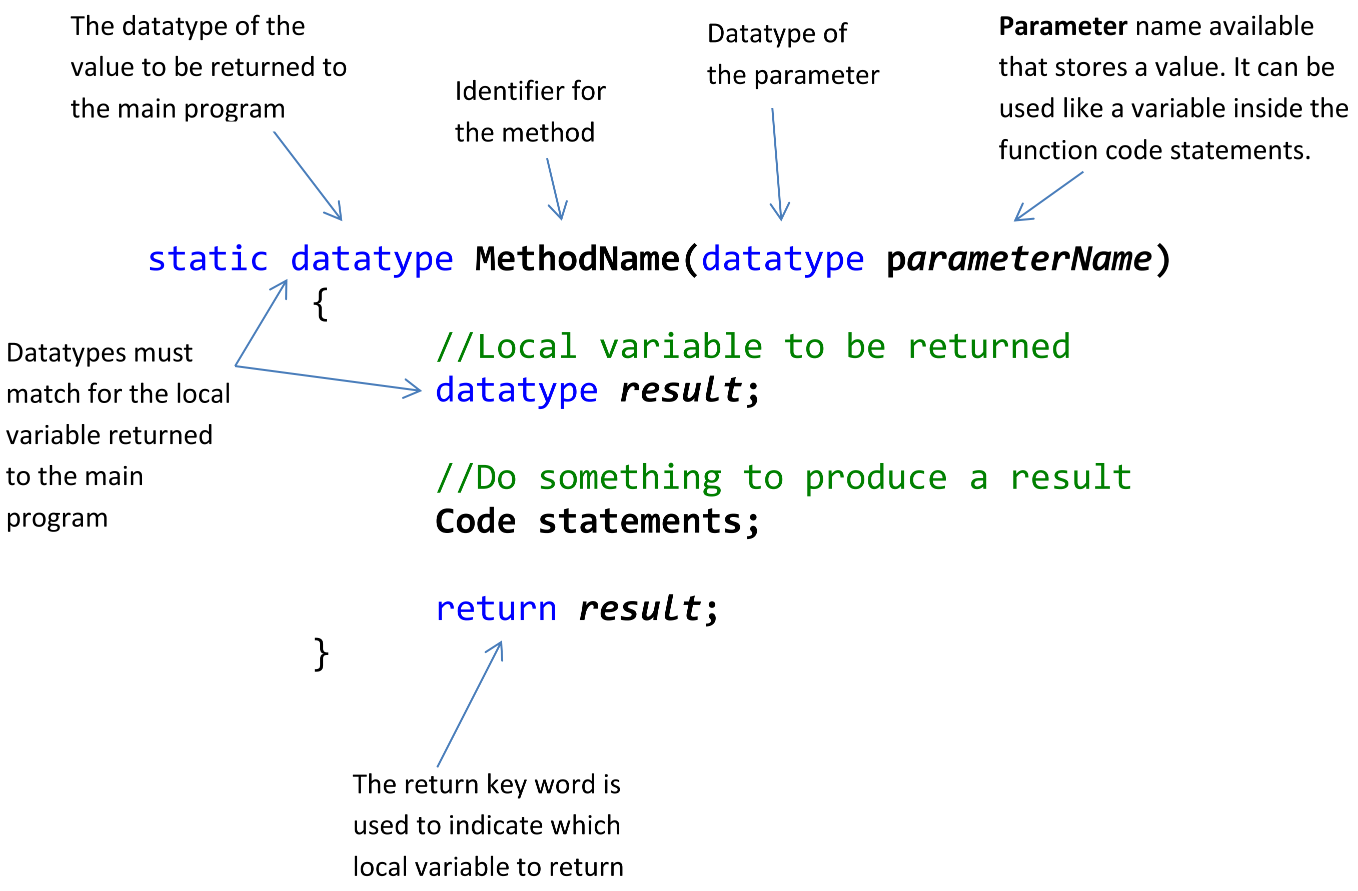
Nope, there is a subtle difference.

A procedure can ***output a value***, like the *number* above, but that ***value is not returned*** to the main program to use later in the main program.

In other words, you cannot take the *number* output to the console and use it as an input to another calculation or subroutine. This is when functions are used.

FUNCTION

A **function** is a type of subroutine that performs a task that ***always*** returns a value to the main program. It is common for a function to perform a calculation.



```
static datatype MethodName(datatype parameterName)
                {
                        //Local variable to be returned
                        datatype result;

                        //Do something to produce a result
                        Code statements;

                        return result;
                }
```

The datatype of the value to be returned to the main program
Identifier for the method
Datatype of the parameter
Parameter name available that stores a value. It can be used like a variable inside the function code statements.
Datatypes must match for the local variable returned to the main program
The return key word is used to indicate which local variable to return

As you can see there are quite a few parts to keep an eye on, how about a handy checklist to help:

1	You have a sensible name for the function	
2a	You know what datatype is needed for the return value e.g. string, int, double	
2b	You have a local return variable with the same datatype	
3a	You have a sensible parameter name	
3b	You have the correct datatype that matches the parameter value being input	
4	Your code statements use the correct parameter name(s)	
5a	You have a return statement	
5b	The return statement uses the correct local variable	
6	You have used the correct syntax	

If everything above looks correct, check the logic of your code!

Function example:

The following is a simple function that adds 10 to any age input.

```
static int CalcFutureAge(int currentAge)
     {
          //Local variable
          int result;

          //Add 10 to value passed into currentAge
          parameter
          result = currentAge + 10;

          //Return result local variable
          return result;
     }
```

Function example with two parameters:

```
static double CalcArea(double length, double width)

    {
          //Local variable
          Double totalArea;

          //Do something to produce a result
          totalArea = length*width;

          return totalArea;
    }
```

CALLING A METHOD

So far you have seen examples of methods (procedures and functions) but not how to use them.

Until you ask them to run in the main section any user-defined methods (ones you have made) will just sit their looking pretty. The next step is to call them.

I like to think of using a method like asking a friend to do something for you. But first you have to give them a call.

C# STRUCTURE

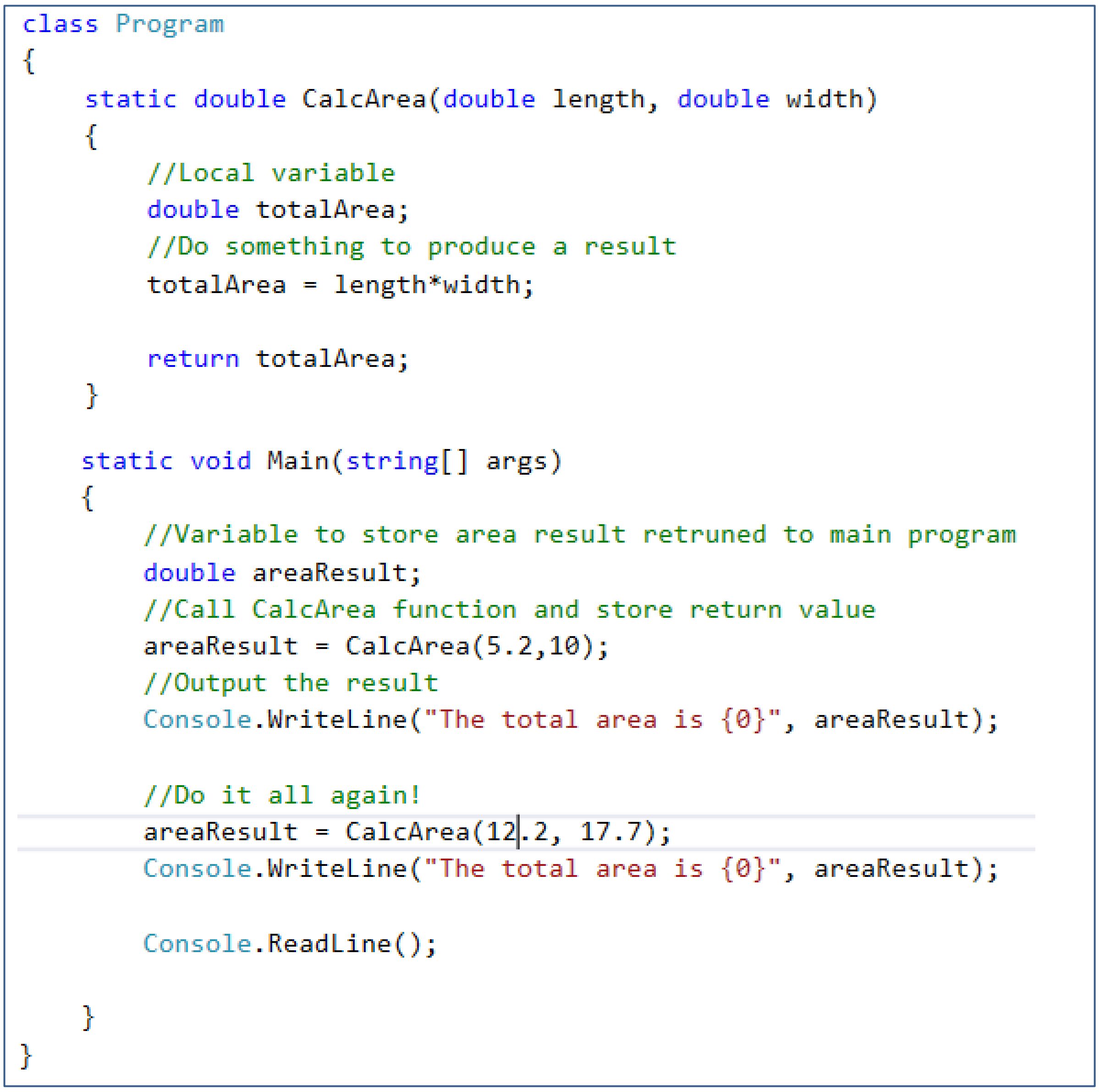
```
static void MyProcedure()
{
      Code statements;
}

static datatype MyFunction(datatype myParam)
{
      //Local variable to be returned
      datatype result;

      //Do something to produce a result
      Code statements;

      return result;
}


static void Main(string[] args)
{
      //Global variable to store function result
      datatype returnResult;

      //Call the procedure to run it
      MyProcedure();

      //Call the function to run it
      returnResult = MyFunction(value)
      //The return value is stored in a variable for
      later use

}
```

The value to be used in the function is input

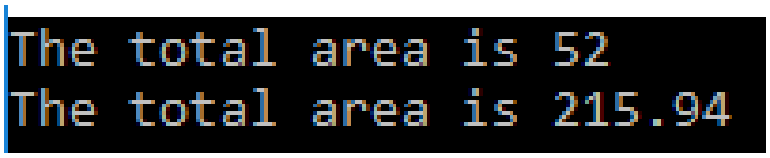
It ***must*** be the correct datatype

CALLING A METHOD EXAMPLE

This example calculates an area total using a function and then does it again using the same function with different input values being passed.



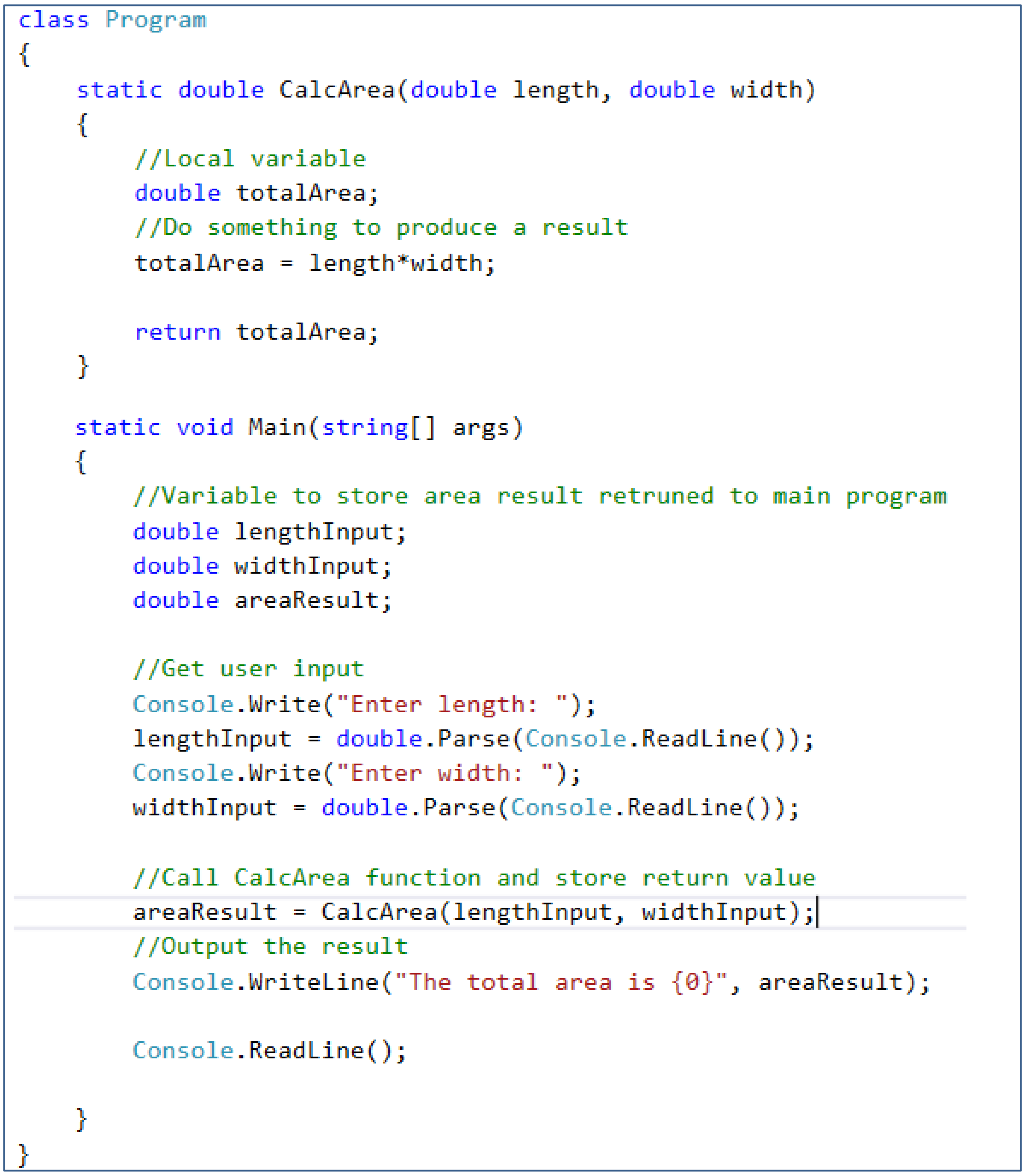
```
class Program
{
    static double CalcArea(double length, double width)
    {
        //Local variable
        double totalArea;
        //Do something to produce a result
        totalArea = length*width;

        return totalArea;
    }

    static void Main(string[] args)
    {
        //Variable to store area result retruned to main program
        double areaResult;
        //Call CalcArea function and store return value
        areaResult = CalcArea(5.2,10);
        //Output the result
        Console.WriteLine("The total area is {0}", areaResult);

        //Do it all again!
        areaResult = CalcArea(12.2, 17.7);
        Console.WriteLine("The total area is {0}", areaResult);

        Console.ReadLine();

    }
}
```



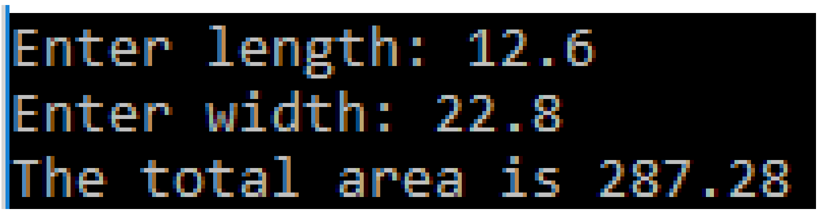
```
The total area is 52
The total area is 215.94
```

CALLING A METHOD EXAMPLE

This is the same example, but this time the user is asked to input values and the global variables used are passed into the function as the arguments into the parameters.



```csharp
class Program
{
    static double CalcArea(double length, double width)
    {
        //Local variable
        double totalArea;
        //Do something to produce a result
        totalArea = length*width;

        return totalArea;
    }

    static void Main(string[] args)
    {
        //Variable to store area result retruned to main program
        double lengthInput;
        double widthInput;
        double areaResult;

        //Get user input
        Console.Write("Enter length: ");
        lengthInput = double.Parse(Console.ReadLine());
        Console.Write("Enter width: ");
        widthInput = double.Parse(Console.ReadLine());

        //Call CalcArea function and store return value
        areaResult = CalcArea(lengthInput, widthInput);
        //Output the result
        Console.WriteLine("The total area is {0}", areaResult);

        Console.ReadLine();

    }
}
```

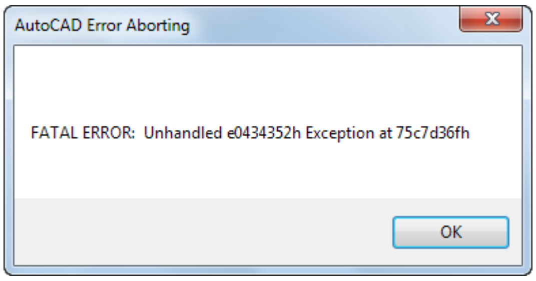


```
Enter length: 12.6
Enter width: 22.8
The total area is 287.28
```

EXCEPTION HANDLING

An exception is when a problem occurs when the program is running (executing), so the program is running along fine and then, BAM! a problem occurs.

At this point:


AutoCAD Error Aborting

FATAL ERROR: Unhandled e0434352h Exception at 75c7d36fh

Option A) your program could crash, with a fatal error

OR

Option B) your program can anticipate there might be a problem, catch it and handle it, no drama, no fuss.

OPTION B

Option B is the way to go but unfortunately without experience anticipating problems can be difficult, which is why testing is so important. Testing allows you to find out where exception handling is needed. A good programmer will try and handle potential problems and in an ideal world no program would ever crash.

TRY-CATCH-FINALLY

The try-catch-finally statement controls what happens when an error occurs while your programming is running.

C# STRUCTURE

```
try
{
     Try running some code statements;
}

catch (type of error errorMessage)
{
     Run these code statements if there is an error
     that matches the type;

     errorMessage is an optional variable used to
     store the message thrown by the program, it is
     often shortened to e
}

finally
{
     Whatever happens always run these code
     statements;
}
```

Note: There can be more than one catch code block, one for each error type

TESTING TO CATCH ANY ERROR

When writing a program is sometimes difficult to know what errors might occur, it is always better to provide a specific message for a particular type of error if you can.

Here is a way to catch-all errors, whatever the type.

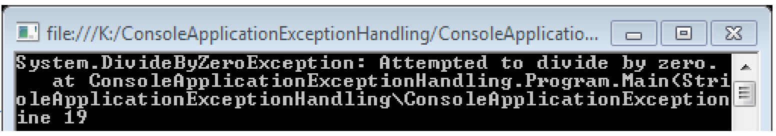
C# STRUCTURE

```
try
{
        Code statements;
}

catch (Exception e)
{
        //Display any error message in the e variable
        Console.WriteLine(e);
}

finally
{
        Code statements;
}
```

C# EXAMPLE

In this example there is a divide by zero error:

```
static void Main(string[] args)
{
    int number, divisor, result;

    try
    {
        number = 12;
        divisor = 0;
        result = number / divisor;
    }
    catch(Exception e)
    {
        //Display error
        Console.WriteLine(e);
    }
    finally
    {
        //Pause console window
        Console.Read();
    }
```


file:///K:/ConsoleApplicationExceptionHandling/ConsoleApplicatio...

```
System.DivideByZeroException: Attempted to divide by zero.
   at ConsoleApplicationExceptionHandling.Program.Main(Stri
oleApplicationExceptionHandling\ConsoleApplicationException
ine 19
```

The error message gives some information that can be used to improve your code. You can see it is a DivideByZeroError

HANDLING SPECIFIC ERRORS

There are many different types of error that can happen, if possible it is better to give specific feedback to a user so they know what to do next

COMMON ERROR TYPES

Here are some of the most common errors that you might encounter

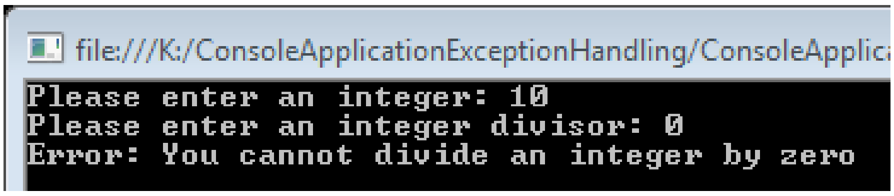
SYSTEM.DIVIDEBYZEROEXCEPTION

Handles errors generated from dividing a dividend with zero.

C# EXAMPLE

Here is an example with a user input.

```
int number, divisor, result;

try
{
    Console.Write("Please enter an integer: ");
    number = int.Parse(Console.ReadLine());

    Console.Write("Please enter an integer divisor: ");
    divisor = int.Parse(Console.ReadLine());
    result = number / divisor;

    Console.WriteLine("Result of {0}/{1} is {2}", number, divisor, result);
}
//Divide by zero error
catch (System.DivideByZeroException e)
{
    //Display error message
    Console.WriteLine("Error: You cannot divide an integer by zero");
}
finally
{
    //Pause console window
    Console.Read();
}
```


file:///K:/ConsoleApplicationExceptionHandling/ConsoleApplic

```
Please enter an integer: 10
Please enter an integer divisor: 0
Error: You cannot divide an integer by zero
```

SYSTEM.FORMATEXCEPTION

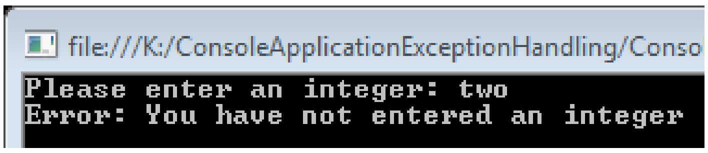
Handles errors generated during casting (changing) data to another format, it is thrown when trying to perform some type of conversion an invalid type e.g. a string to an integer

C# EXAMPLE

Here is an example with a user input, where a user enters words instead of an integer.

```
int number, divisor, result;

try
{
    Console.Write("Please enter an integer: ");
    number = int.Parse(Console.ReadLine());

    Console.Write("Please enter an integer divisor: ");
    divisor = int.Parse(Console.ReadLine());
    result = number / divisor;

    Console.WriteLine("Result of {0}/{1} is {2}", number, divisor, result);
}
//Type conversion error
catch (System.FormatException e)
{
    //Display error message
    Console.WriteLine("Error: You have not entered an integer");
}
finally
{
    //Pause console window
    Console.Read();
}
```

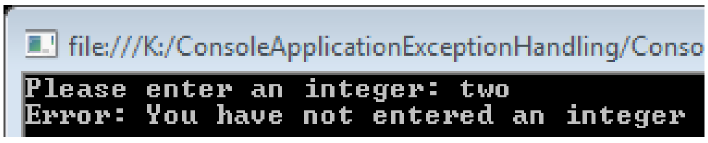
Test for entering “two” as the integer



file:///K:/ConsoleApplicationExceptionHandling/Conso

```
Please enter an integer: two
Error: You have not entered an integer
```

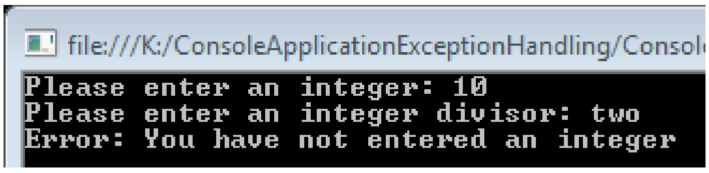
Test for entering no value



file:///K:/ConsoleApplicationExceptionHandling/Conso

```
Please enter an integer: two
Error: You have not entered an integer
```

Test for entering “two” as the divisor



file:///K:/ConsoleApplicationExceptionHandling/Consol

```
Please enter an integer: 10
Please enter an integer divisor: two
Error: You have not entered an integer
```

SYSTEM.INDEXOUTOFRANGEEXCEPTION

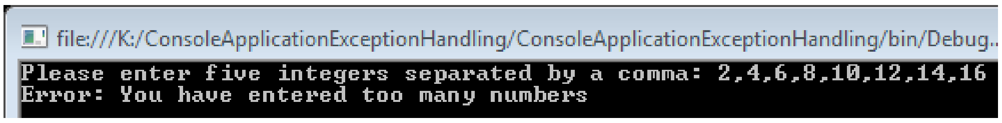
Use when working with an array, it handles errors when an array index out of range.

C# EXAMPLE

This example program asks for five numbers, loops through the input string and converts each number to an integer, then stores it in the array before doing some division on each number.

```csharp
//Setup array and variables
string newNumbers;
int[] numbers = new int[5];
int divisor, result;

try
{
    Console.Write("Please enter five integers separated by a comma: ");
    newNumbers = Console.ReadLine();

    //Split into an array
    string[] splitNumbers = newNumbers.Split(',');

    //Loop through each item in the new array and convert into an int
    //Add parsed value into
    int index = 0;
    foreach (string item in splitNumbers)
    {
        numbers[index] = int.Parse(item);
        index++;
    }

    //Get the divisor
    Console.Write("Please enter an integer divisor: ");
    divisor = int.Parse(Console.ReadLine());

    //Divide each number and output the result
    for (int i = 0; i < 5; i++)
    {
        result = numbers[i] / divisor;
        Console.WriteLine("Result of {0}/{1} is {2}", numbers[i], divisor, result);
    }

}
//Index out of range error
catch (System.IndexOutOfRangeException e)
{
    //Display error message
    Console.WriteLine("Error: You have entered too many numbers");
}
finally
{
    //Pause console window
    Console.Read();
}
```

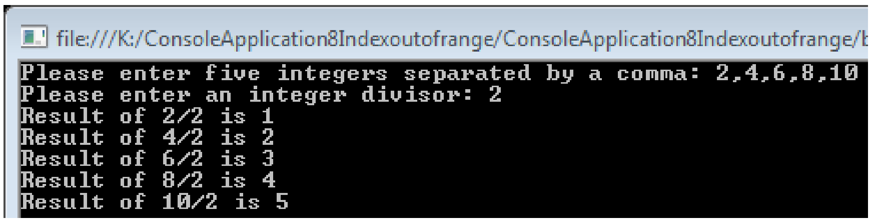
Testing with too many numbers



file:///K:/ConsoleApplicationExceptionHandling/ConsoleApplicationExceptionHandling/bin/Debug..

```
Please enter five integers separated by a comma: 2,4,6,8,10,12,14,16
Error: You have entered too many numbers
```

Testing with five numbers



file:///K:/ConsoleApplication8Indexoutofrange/ConsoleApplication8Indexoutofrange/b

```
Please enter five integers separated by a comma: 2,4,6,8,10
Please enter an integer divisor: 2
Result of 2/2 is 1
Result of 4/2 is 2
Result of 6/2 is 3
Result of 8/2 is 4
Result of 10/2 is 5
```

ROBUST SUBROUTINE CODE

All of the examples only handle one type of error at time, what you really need is to be able to handle if any of those errors occur and even unexpected ones.

COMBINING WITH FUNCTIONS

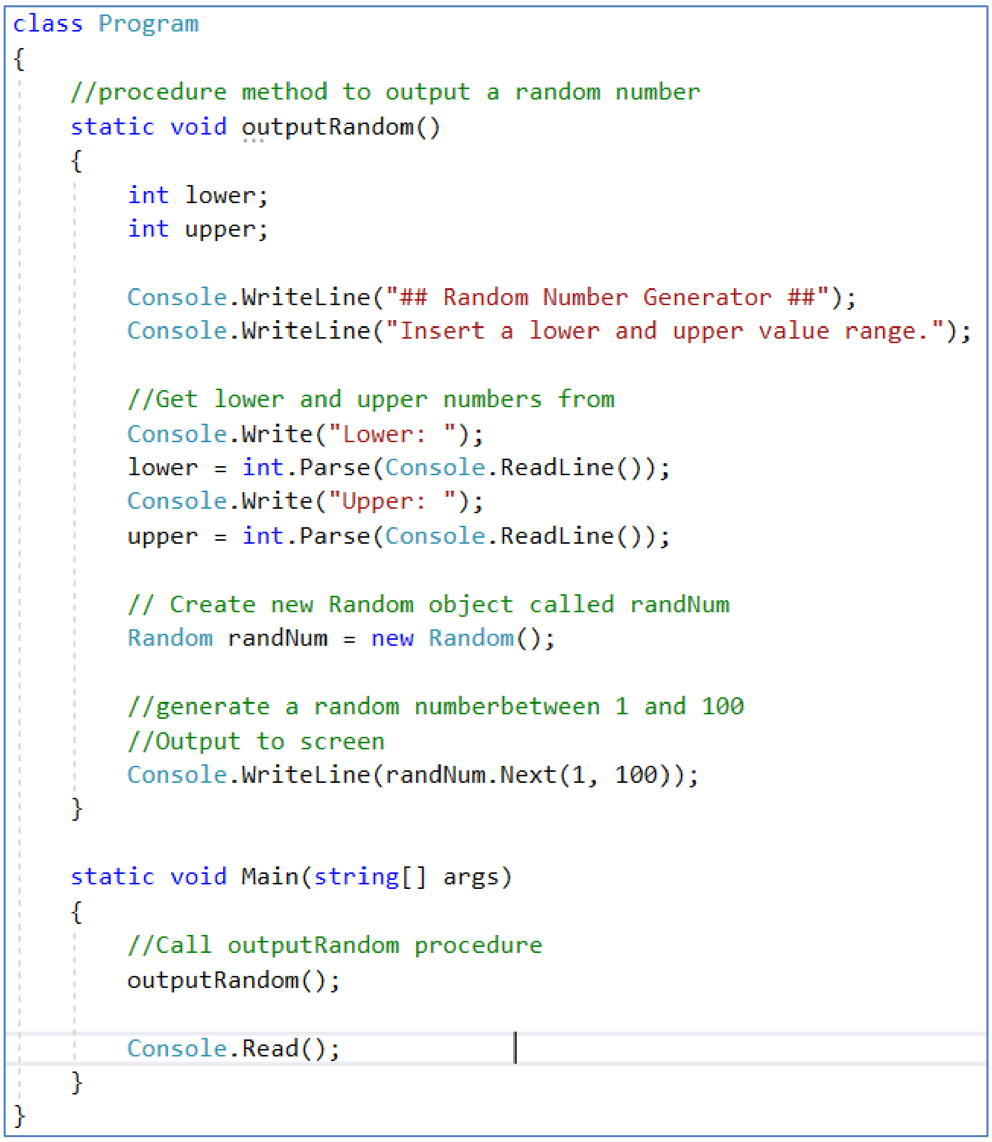
Subroutines (functions /procedures) are a very good way to structure your code but ideally they should also check for errors to make them more robust. Which means it doesn't crash so easily!

C# EXAMPLE - OUTPUT A RANDOM NUMBER

This example is a procedure that outputs a random number; This could be written in many different, with input parameters, as a function with a return value, it depend what you are trying to achieve.

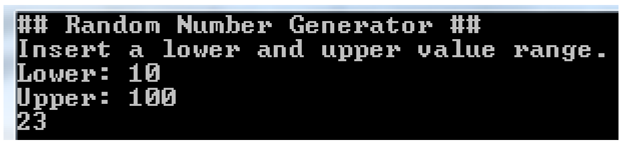
NOT ROBUST

In this example, a user is asked to input two numbers and a random number is generated and displayed. But if a user enters a non-number value it will error and crash.



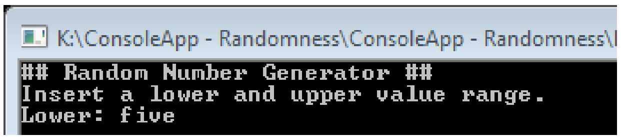
```
class Program
{
    //procedure method to output a random number
    static void outputRandom()
    {
        int lower;
        int upper;

        Console.WriteLine("## Random Number Generator ##");
        Console.WriteLine("Insert a lower and upper value range.");

        //Get lower and upper numbers from
        Console.Write("Lower: ");
        lower = int.Parse(Console.ReadLine());
        Console.Write("Upper: ");
        upper = int.Parse(Console.ReadLine());

        // Create new Random object called randNum
        Random randNum = new Random();

        //generate a random numberbetween 1 and 100
        //Output to screen
        Console.WriteLine(randNum.Next(1, 100));
    }

    static void Main(string[] args)
    {
        //Call outputRandom procedure
        outputRandom();

        Console.Read();
    }
}
```

Example with correct values input



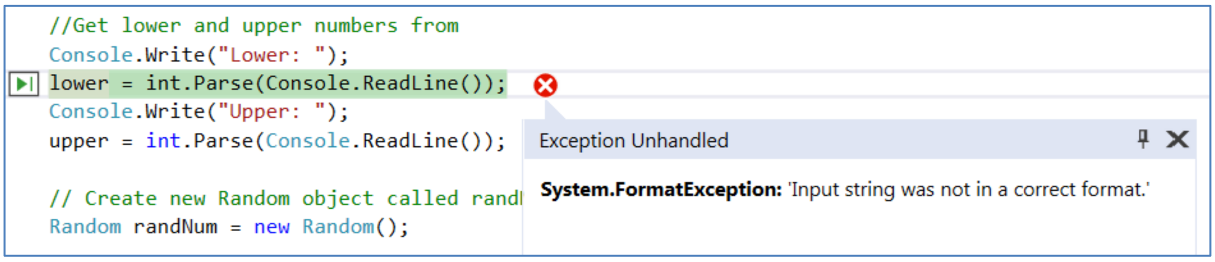
```
## Random Number Generator ##
Insert a lower and upper value range.
Lower: 10
Upper: 100
23
```

Example with incorrect values input



```
K:\ConsoleApp - Randomness\ConsoleApp - Randomness\
## Random Number Generator ##
Insert a lower and upper value range.
Lower: five
```

Entering ‘five’ causes an exception and the program crashes when the code tries to convert ‘five’ into an integer. Whoops!



```
//Get lower and upper numbers from
Console.Write("Lower: ");
lower = int.Parse(Console.ReadLine());
Console.Write("Upper: ");
upper = int.Parse(Console.ReadLine());

// Create new Random object called rand
Random randNum = new Random();
```

Exception Unhandled

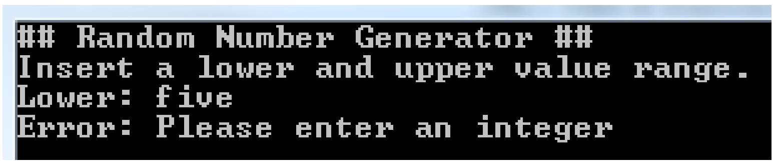
System.FormatException: 'Input string was not in a correct format.'

ROBUST

Again, in this example, a user is asked to input two numbers and a random number is generated and displayed. This time if a user enters a non-number value the code will handle the exception and the program won’t crash.

```
//procedure method to output a random number
static void outputRandom()
{
    int lower;
    int upper;

    Console.WriteLine("## Random Number Generator ##");
    Console.WriteLine("Insert a lower and upper value range.");

    //Try to get a number
    try
    {
        //Get lower and upper numbers from
        Console.Write("Lower: ");
        lower = int.Parse(Console.ReadLine());
        Console.Write("Upper: ");
        upper = int.Parse(Console.ReadLine());

        // Create new Random object called randNum
        Random randNum = new Random();

        //generate a random numberbetween 1 and 100
        //Output to screen
        Console.WriteLine(randNum.Next(1, 100));
    }
    //Catch if int.Parse has a probelm with a non-number
    catch(System.FormatException)
    {
        //Display error message
        Console.WriteLine("Error: Please enter an integer");
    }
}
```

Example with incorrect values input



```
## Random Number Generator ##
Insert a lower and upper value range.
Lower: five
Error: Please enter an integer
```

Still more robust code

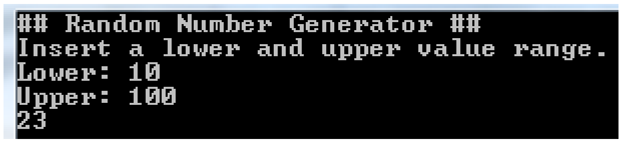
Well it works but stops if a user enters an incorrect value. Often that isn't enough, you want the user to keep trying until they enter the correct information.

This is where a while loop can be useful; basically keep going while the user gets it wrong!

Pseudocode:

```
#A variable is needed to control the WHILE loop
noErrors ←False

WHILE noErrors NOT EQUAL TO True THEN
    TRY:
        #All lines of code to try are listed here
        Code…..
        Code…..
        Code…..

        #If there are no errors, the code reaches this point
        #success changes to True and while loop stops
        noErrors ←True

    CATCH EXCEPTION:
        #If there are any errors when executing the code
        OUTPUT "There has been an error, try again"
ENDWHILE
```

Example with correct values input

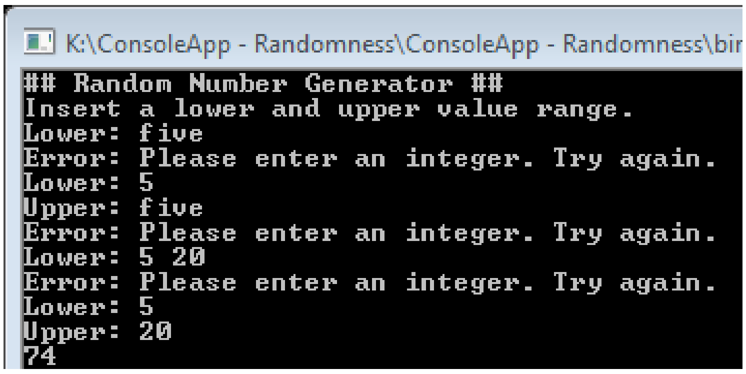


```
## Random Number Generator ##
Insert a lower and upper value range.
Lower: 10
Upper: 100
23
```

Example with incorrect values input

Wow, it keeps going!

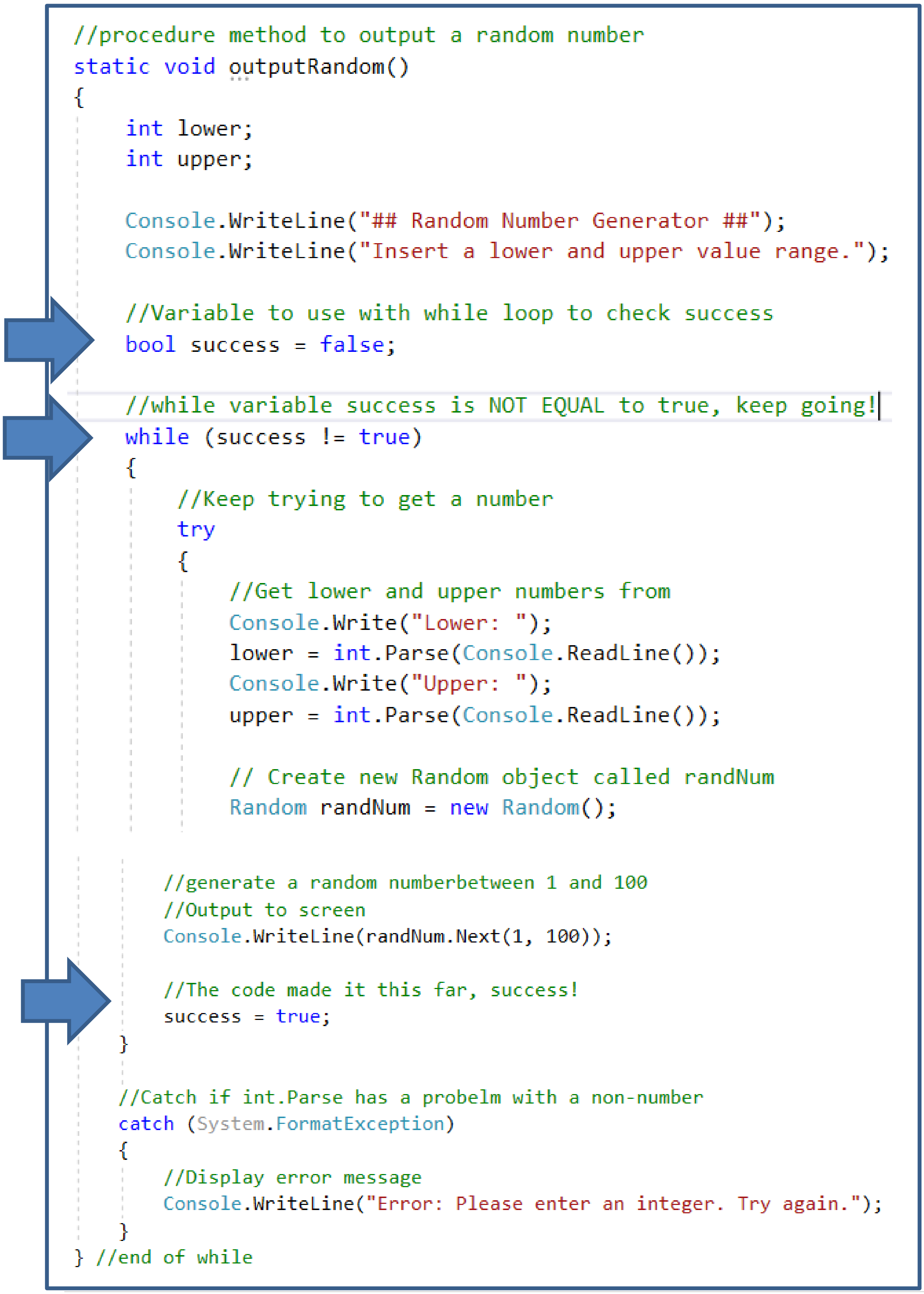
See the code on the next page.



K:\ConsoleApp - Randomness\ConsoleApp - Randomness\bir

```
## Random Number Generator ##
Insert a lower and upper value range.
Lower: five
Error: Please enter an integer. Try again.
Lower: 5
Upper: five
Error: Please enter an integer. Try again.
Lower: 5 20
Error: Please enter an integer. Try again.
Lower: 5
Upper: 20
74
```

Still more robust code example – While and Try



```
//procedure method to output a random number
static void outputRandom()
{
    int lower;
    int upper;

    Console.WriteLine("## Random Number Generator ##");
    Console.WriteLine("Insert a lower and upper value range.");

    //Variable to use with while loop to check success
    bool success = false;

    //while variable success is NOT EQUAL to true, keep going!
    while (success != true)
    {
        //Keep trying to get a number
        try
        {
            //Get lower and upper numbers from
            Console.Write("Lower: ");
            lower = int.Parse(Console.ReadLine());
            Console.Write("Upper: ");
            upper = int.Parse(Console.ReadLine());

            // Create new Random object called randNum
            Random randNum = new Random();

        //generate a random numberbetween 1 and 100
        //Output to screen
        Console.WriteLine(randNum.Next(1, 100));

        //The code made it this far, success!
        success = true;
    }

    //Catch if int.Parse has a probelm with a non-number
    catch (System.FormatException)
    {
        //Display error message
        Console.WriteLine("Error: Please enter an integer. Try again.");
    }
} //end of while
```

PUTTING IT ALL TOGETHER – COMPLETE CODE EXAMPLE

There is room for improvement in the game, such as adding a scoring system or breaking into further subroutines. Feel free to experiment and improve.

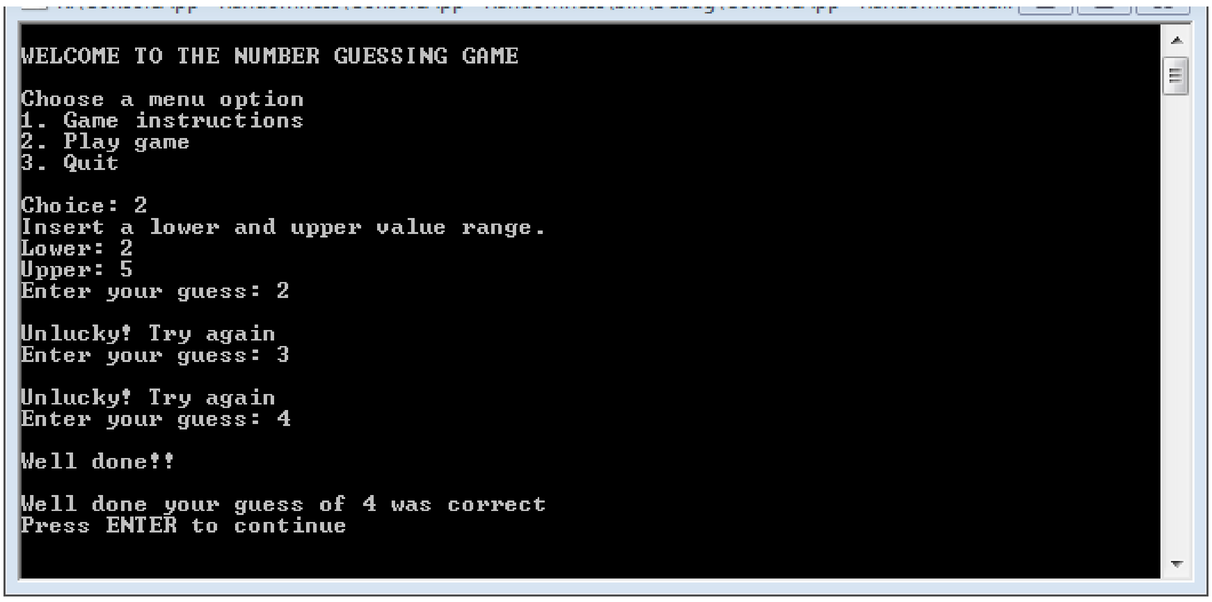
When coding, one aim is to limit the use of global variables as they have an effect on memory usage. Another aim is to use subroutines to solve each requirement for the code, this makes the code much easier to update and maintain.

NUMBER GUESSING GAME

The following code is a simple number guessing game; it has a menu of options for a user to choose from. A user chooses the range of numbers (lower to upper) and then a random number is generated. The user then tries to guess the correct number.

The code includes validation of menu choice and error handling if an incorrect value type is entered that cannot be converted into an integer, for example, five instead of 5.

Here is an example of the game working with a simple test.



```
WELCOME TO THE NUMBER GUESSING GAME

Choose a menu option
1. Game instructions
2. Play game
3. Quit

Choice: 2
Insert a lower and upper value range.
Lower: 2
Upper: 5
Enter your guess: 2

Unlucky! Try again
Enter your guess: 3

Unlucky! Try again
Enter your guess: 4

Well done!!

Well done your guess of 4 was correct
Press ENTER to continue
```

SUBROUTINES

There are a number of self-contained subroutines that are used, each use their own local variables, which are a much more efficient use of memory.

GetMenuChoice()

- Uses a while loop to validate the menu choice. The code repeats until the user enters a valid choice.
    - Displays the menu of options by calling ***DisplayMenu()***
    - Catches an input exception if the user enters a incorrect value
        - The int.Parse() will cause an exception error if an incorrect value is entered that cannot be converted to an integer e.g. two, instead of 2.
- The local variable ***choice*** is returned to the main program.

DisplayMenu()

- Displays the menu options and is called from ***GetMenuChoice()***

Getrandom()

- Is called when user chooses to play the game.
- Asks the user to enter a lower and upper range of numbers.
    - This is not currently validated to see if the numbers entered are correct i.e. upper is bigger than lower.
- Uses a while loop to validate the values are integers. The code repeats until the user enters a valid integer type.
    - Catches an input exception if the user enters a incorrect value type e.g. one hundred, instead of 100
- The local variable ***randNum*** is returned to the main program to be passed into ***CorrectGuess(***<u>*randNum*</u>***,*** *guess****)***
-
    - Note a local variable needs to be given an initial value, otherwise you will get a syntax error
    - Also, you will get a syntax error if you place the return statement in the wrong place.
        - A function should always return a value, for every path in the code. This is most commonly a problem with IF and WHILE statements. Be careful.

GetGuess()

- Is called when user chooses to play the game.
- Asks the user to enter a guess for the answer.
- Uses a while loop to validate the guess is an integers. The code repeats until the user enters a valid integer type.
    - Catches an input exception if the user enters a incorrect value type
- The local variable ***guess*** is returned to the main program to be passed into ***CorrectGuess(****randNum****,*** <u>*guess*</u>***)***

CorrectGuess(*randNum****,* *guess****)***

- Is called when user chooses to play the game.
- Compares the two parameter values of **randNum** and **guess**.
- If the value match, sets the local variable ***correct*** to true and displays a message.
- Otherwise the ***correct*** remains as false and an alternative message is displayed.
- The local variable ***correct*** is returned to the main program and is used in a while loop in the main code.
  - The while loop repeatedly calls the ***GetGuess() and CorrectGuess(****randNum****,* *guess****)*** until the return value of correct is true.

MAIN

This is where all the code is controlled and subroutines are called. This is the key sequence of code and controls the logic of the program.

Main - Global variables

There are four global variables ***choice, randNum*** , ***guess*** and ***correct***.

- ***choice*** is returned from the ***GetMenuChoice()*** subroutine and is used the main while loop that controls when the game is finished.

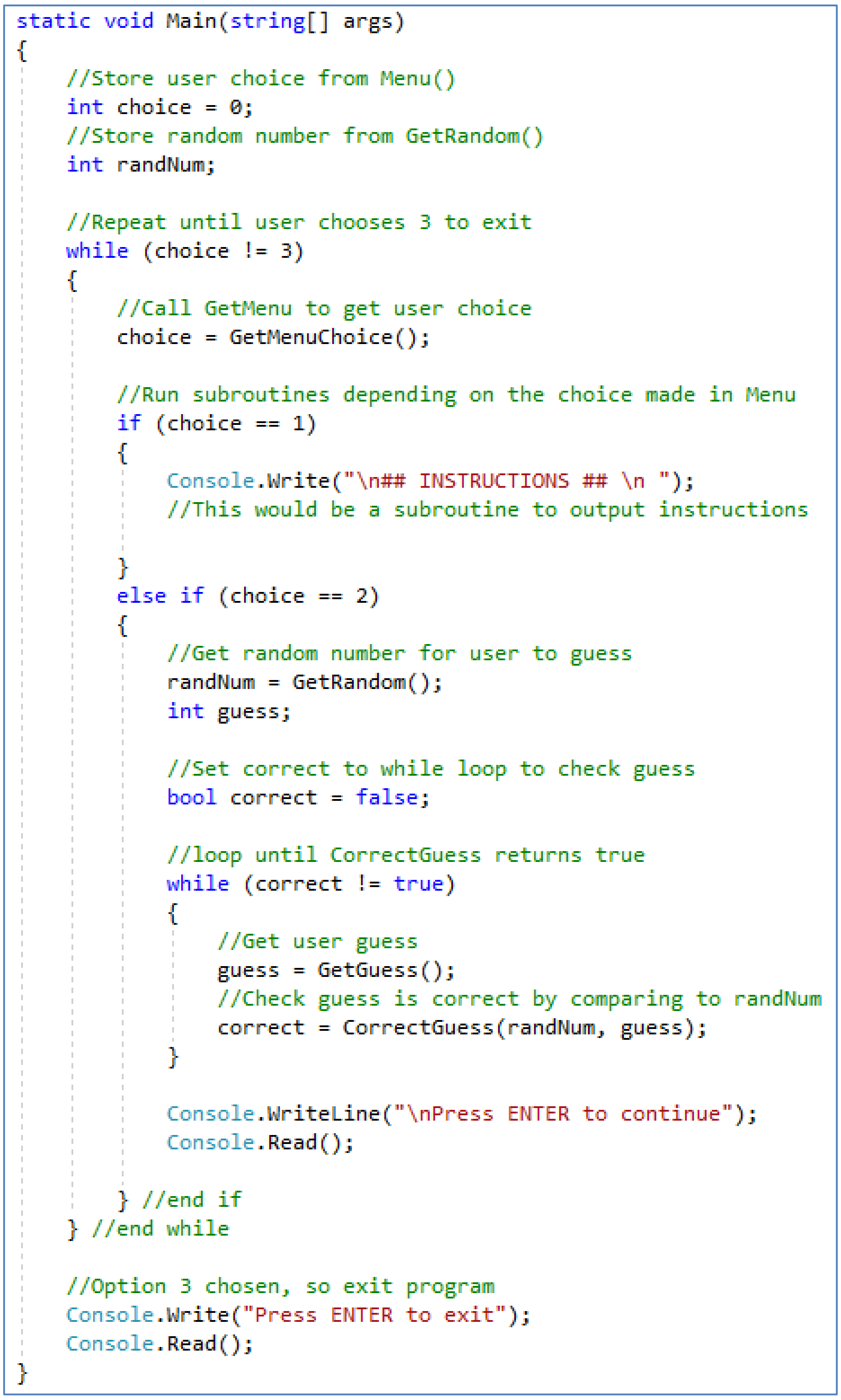
- ***randNum*** is returned from the ***GetRandom()*** subroutine, which generates a random number for the game.
  - The value stored in ***randNum*** is passed into the ***CorrectGuess(<u>randNum</u>, guess)*** subroutine to compare with ***guess***.

- ***guess*** is returned from the ***GetGuess()*** subroutine, which asks a user to input a guess value.
  - The value stored in ***guess*** is passed into the ***CorrectGuess(randNum, <u>guess</u>)*** subroutine to compare with ***randNum***.

- ***correct*** is returned from the ***CorrectGuess(randNum, guess)*** subroutine, which is used in a while loop that repeats until a correct guess has been made.
  - Once a correct guess is made the game has finished and the menu is displayed to see if the user wants to play again.

C# CODE

Main



```
static void Main(string[] args)
{
    //Store user choice from Menu()
    int choice = 0;
    //Store random number from GetRandom()
    int randNum;

    //Repeat until user chooses 3 to exit
    while (choice != 3)
    {
        //Call GetMenu to get user choice
        choice = GetMenuChoice();

        //Run subroutines depending on the choice made in Menu
        if (choice == 1)
        {
            Console.Write("\n## INSTRUCTIONS ## \n ");
            //This would be a subroutine to output instructions

        }
        else if (choice == 2)
        {
            //Get random number for user to guess
            randNum = GetRandom();
            int guess;

            //Set correct to while loop to check guess
            bool correct = false;

            //loop until CorrectGuess returns true
            while (correct != true)
            {
                //Get user guess
                guess = GetGuess();
                //Check guess is correct by comparing to randNum
                correct = CorrectGuess(randNum, guess);
            }

            Console.WriteLine("\nPress ENTER to continue");
            Console.Read();

        } //end if
    } //end while

    //Option 3 chosen, so exit program
    Console.Write("Press ENTER to exit");
    Console.Read();
}
```

GetMenuChoice()

```
//Get user menu choice
    //int indicates an integer return value
static int GetMenuChoice()
{
    //variable for user choice, starts while loop
    //Choice is an int so it can be validated easily
    int choice = 0;

    //Check valid choice
    //Repeat until valid user choice
    while (choice < 1 || choice > 3)
    {
        //Call DisplayMenu
        DisplayMenu();

        //Try code that might cause an error
        try
        {
            Console.Write("Choice: ");
            choice = int.Parse(Console.ReadLine()); //get choice, convert into int
        }
        //Catch errors, if doesn't enter an integer
        catch (System.FormatException)
        {
            //Display error message
            Console.WriteLine("Error: Please enter an integer. Try again.");
        }

    }
    //Valid choice, return to main program once while finished
    return choice;
}
```

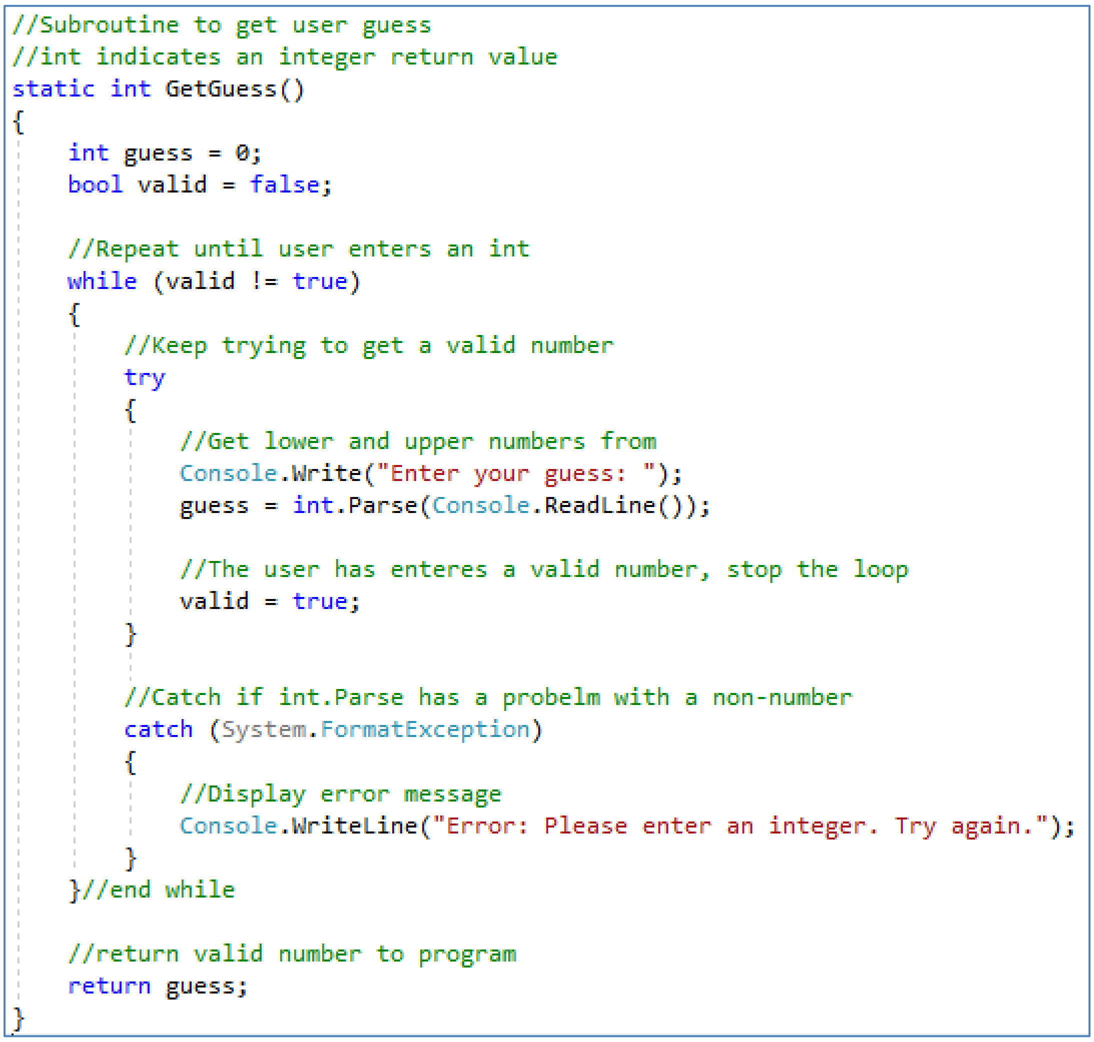
DisplayMenu()

```
//Menu for displaying options
    //void indicates no return value
static void DisplayMenu()
{
    //Output choices
    Console.WriteLine("\nWELCOME TO THE NUMBER GUESSING GAME \n");
    Console.WriteLine("Choose a menu option");
    Console.WriteLine("1. Game instructions");
    Console.WriteLine("2. Play game");
    Console.WriteLine("3. Quit \n");
}
```

Getrandom()

```
//Subroutine to generate a random number
    //int indicates an integer return value
static int GetRandom()
{
    int lower;
    int upper;
    int randNum = 0; //must initialise a local variable a value that is returned

    Console.WriteLine("Insert a lower and upper value range.");

    //Variable to use with while loop to check success
    bool valid = false;

    //while variable success is NOT EQUAL to true, keep going!
        //Keep trying to get an integer
    while (valid != true)
    {
        //Try code that might cause an error
        //Converting user input to an int
        try
        {

            //Get lower and upper numbers from
            Console.Write("Lower: ");
            lower = int.Parse(Console.ReadLine());
            Console.Write("Upper: ");
            upper = int.Parse(Console.ReadLine());

            // Create new Random object called randNum
            Random newNum = new Random();

            //generate a random numberbetween 1 and 100
            //store in randNum variable to return
            randNum = newNum.Next(lower, upper);

            //The code made it this far, success!
            valid = true; //could use break instead to stop the loop
        }

        //Catch if int.Parse has a probelm with a non-number
        catch (System.FormatException)
        {
            //Display error message
            Console.WriteLine("Error: Please enter an integer. Try again.");
        }

    } //end of while

    //Make sure that a value is always returned after while loop stops
    return randNum;
}
```

GetGuess()



```csharp
//Subroutine to get user guess
//int indicates an integer return value
static int GetGuess()
{
    int guess = 0;
    bool valid = false;

    //Repeat until user enters an int
    while (valid != true)
    {
        //Keep trying to get a valid number
        try
        {
            //Get lower and upper numbers from
            Console.Write("Enter your guess: ");
            guess = int.Parse(Console.ReadLine());

            //The user has enteres a valid number, stop the loop
            valid = true;
        }

        //Catch if int.Parse has a probelm with a non-number
        catch (System.FormatException)
        {
            //Display error message
            Console.WriteLine("Error: Please enter an integer. Try again.");
        }
    }//end while

    //return valid number to program
    return guess;
}
```

CorrectGuess(*randNum, guess****)***

```
//Compare random number and guess, return true if correct
    //bool indicates an boolean (true/false) return value
static bool CorrectGuess(int randNum, int guess)
{
    bool correct = false;

    //Compare numbers
    if (randNum==guess)
    {
        //Correct guess, true
        correct = true;
        Console.WriteLine("\nWell done!!");

        //Display correct message
        Console.Write("\nWell done your guess of {0} was correct", guess);

    }
    else
    {
        Console.WriteLine("\nUnlucky! Try again");
        correct = false;
    }

    //return valid number to program
    return correct;
}
```

RESOURCES

WEBSITES

https://www.tutorialspoint.com/csharp/index.htm

https://www.dotnetperls.com/

https://en.m.wikibooks.org/wiki/C_Sharp_Programming

https://stackoverflow.com/

BOOKS

C# Programming Yellow Book by Rob Miles, University of hull, free download

Learn C# in one day and learn it well by Jamie Chan; ISBN-13: 978-1518800276

C# 6.0 and the .NET 4.6 Framework by Andrew Troelsen; ISBN-13: 978-1484213339

The C# player's Guide by RB Whitaker; ISBN-13: 978-0985580131

VIDEO TUTORIALS

C# tutorial by Derek Banas; YouTube

C# Fundamentals for Absolute Beginners by Bob Tabor; Microsoft Virtual Academy

Programming in C# Jump Start by Jerry Nixon and Daren May; Microsoft Virtual Academy

GAME DEVELOPMENT

Monogame, free cross-platform extension using C#

Unity, advanced cross-platform extension using C#

MOBILE APP DEVELOPMENT

https://www.xamarin.com/ - cross-platform mobile app development with C#