Tutorialr.com

C#

Contents

Introduction

What is Programming?

Programming can seem daunting but if you think about it as another form of writing with its own grammar, style and layout then that's a good way to approach it. From the simplest sample to the most advanced application the concepts are the same, there's just more to write - when reading or writing you have words and paragraphs and programming has its own way being written and breaking up functionality however unlike text you can do many different things with the same piece of functionality if you write it in a way. There are two main rules to help keep programming from being too complicated that are KISS and DRY - which is Keep it Short and Simple and Don't Repeat Yourself with those two things in mind that will help you along the way.

Programming is essentially a set of tasks for a computer to perform and rules for it to follow, this essentially can be a self-contained sequence of actions to be performed - these are known as algorithms and form the basic concept of programming. Think of a sequence of actions like how to make a cup of tea, you can even write this down and that's an algorithm there will be steps to follow, some which need to be done in an order, also maybe different choices such as taking milk or sugar - things that vary in programming are known as variables such as how many lumps of sugar to add when making tea.

When creating an algorithm, it's important to have all the steps explained - each part of this can be a block of functionality such as fill kettle or pour water - if there's a lot of similar steps together you can group them together, much like a paragraph - in programming these are known as functions but may also be referred to as methods. These functions may be used to perform a small sub-set of tasks and can contain their own variables but it's also possible to supply these functions with values - these are known as parameters and a function can use these parameters to use for its task and some functions can even return a value, for example you can have a function to add two numbers together and return the answer.

Just like in writing there's words you need to use to make it make sense such as verbs and nouns and programming has its own which are often known as key or reserved words - these can dictate what a variable might be, can it just be whole numbers which are known as integers or a piece of text which is known as a string. Key words can also set how functions may be used in programming - in a lot of languages there is a concept known as a class, this is where all the code - which is what makes up programming is grouped together to represent something or some common functionality.

There will be functions that can be used only inside one of these classes - those are known as private, and there are those that can be used both inside and outside the class - these are known as public, both functions and variables can be private and public. There's also a special kind of variable called a property, where you can get a value from them or set their value - these are most useful for parts of a class that you want to use such as if a class were to represent a Car you could have a property for number of doors or one for colour. You can also use classes to help organise code but also in many programming languages the minimum amount of code needed can itself be a class and will often have a main method which will be the start of the code to be used.

When you want to write something you'll use a word processing application or something similar, in programming this is also the case as all programming languages have their preferred programming environment - these are often called Integrated Development Environments or IDEs and these allow you to write your code and also make it easier to read with colourisation of keywords and line numbers to help you find parts you have written, there may be other features to help you find out what things you can type in to access additional functionality or related actions. To perform the actions that have been defined in the code it is compiled, this means it is turned into instructions the computer can understand, some programming languages have an additional layer for this but essentially down at the "bottom" there's code the computer uses to run or execute your actions.

The best way to start programming is to write small examples that use the various features of a programming language and perform familiar actions such as addition or subtraction using the programming language to see how it works, programming languages are very good at doing these kind of actions so they're a good way to get started, then you can build up from there - you probably did something similar when learning to read and write where smaller examples can be used to get started and programming should be no different. The best thing to do is as things get more complicated then break them down - you'll have a few lines of code grouped together in functions just like sentences in paragraphs and don't put too many together so that if there's a mistake you can spot it, just like proof-reading you may need to just need to concentrate on small sections at a time to help find anything wrong.

Also don't worry if something doesn't work the first time around, maybe like the making tea example you forgot to put in a step for filling the kettle, sometimes you can learn a lot more from making a mistake to see what you might need to look out for, in programming you might have a variable you expect to be larger than ten but you never set it to anything larger than nine instead might be the kind of problems you encounter, most mistakes in programming are down to assuming something will be in a given state at a particular time, when you perform actions in your code it will go through various states, by breaking up your code into smaller sections you can examine these states when the code is being executed, often called debugging, to see if things are the way they should be, or not in programming!

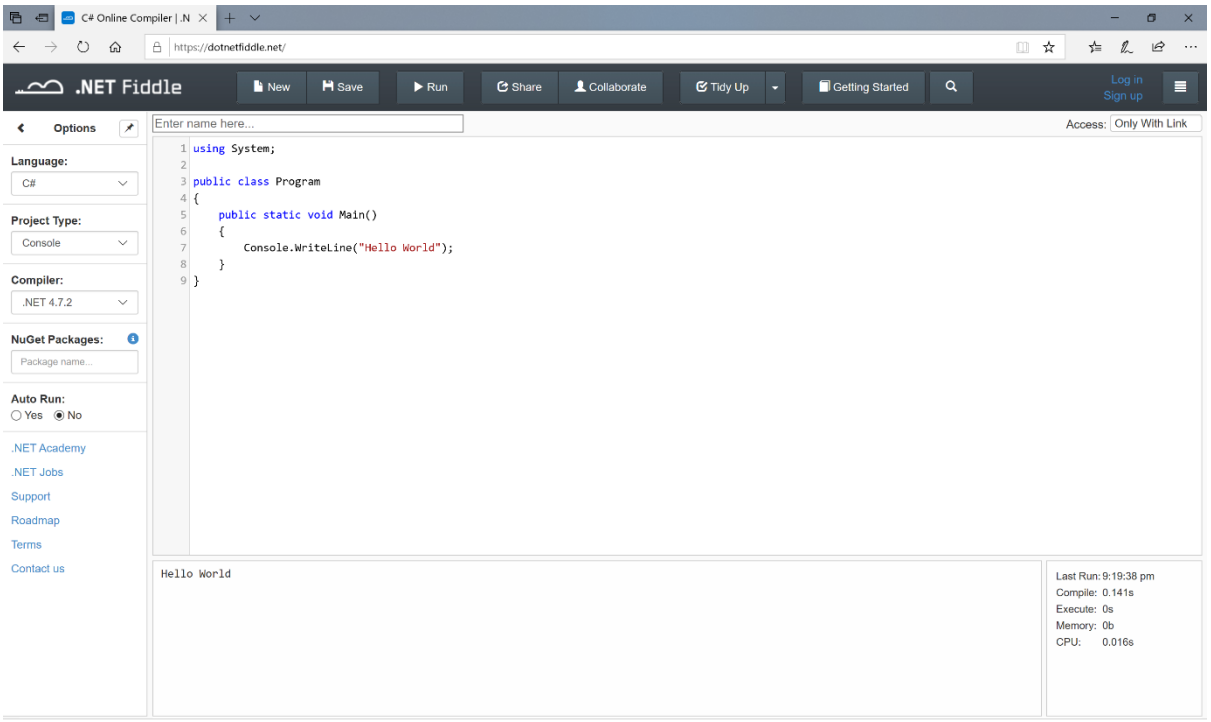
What is C#?

C# is what you'll be learning to write code in - pronounced "C-sharp" like the musical note, this language is like many others although they might have slightly different keywords and other minor differences but belongs to a well-established family of programming languages. C# began as part of the Microsoft .NET Framework - this is one of those layers mentioned previously which executes your code and issues the actual instructions your computer will perform when it runs, it also has all the additional functionality you can take advantage of as part of the framework which contains commonly used and needed functionality so generally you only need to write what you need that's unique to your programme and if you need to do something like open a file then the libraries that make up the .NET Framework are there plus it's possible to use libraries written by other people or companies in your applications but as a beginner you'll just be starting out with the basics but it's nice to know you can take advantage of more functionality should you need it.

C# supports many features common to modern programming languages and has its own structure with blocks of code between curly braces being the most obvious which helps you clearly see where a function or a class starts and ends, it has a variety of keywords you can store the information you need with variables to store the type of information be that integers for whole numbers like 1 to 10 and strings to store text like "Hello World!", each piece of functionality you can use in C# will be introduced and explained along the way but you see many of the things mentioned that are common to all programming which you can apply to other programming languages - but also learn many useful things that you can do in C# to make things easier or to do the actions you need.

What is dotnetfiddle.net?

To run any C# code during this workshop all you need is to be able to access dotnetfiddle.net in your browser, there are two main parts a larger upper text area which will contain the code you need to type in and a lower text area which will show any output:



It will only be necessary to type in any code into the larger upper text area which by default when you first visit dotnetfiddle.net will sow the following:

```
using System;

public class Program
{
	public static void Main()
	{
		Console.WriteLine("Hello World");
	}
}
```

Any output that your code will produce in dotnetfiddle.net will appear in the lower text area, the default example will show the following:

```
Hello World
```

Anything you type will be "run" as you enter it however you can change the "Auto Run" to No if you prefer to select "Run" from the top instead.

Overview

In this workshop we'll start with a small amount of code you'll need to write to get something to happen which is to output something on the screen - this is a good way of seeing the steps of getting something written and working without too much code to go through and work out what it does, anything new will be explained along the way plus various concepts will be repeated and used in different ways just like it's possible to use language in different ways in writing, programming is the same but it's okay to take time to understand something that's new in programming just as it was when learning to read and write.

Also you can always stop and go back over an example and there'll be other concepts you don't see in normal writing such as being able to repeat some actions - called looping or iteration, create decisions or branches and dealing with input and output but always using something that you've learned in a previous part of the workshop should help, you'll not only learn more about programming in general but also how to use the tools and features of a programming language used by millions of people all around the world used in portable devices, to complex business processes, major websites and software applications many of which you've probably seen or used.

C# will be introduced in this workshop in phases with many of the core features you'll need to create small or complex applications introduced, so anything new will be explained along the way so you can always go back later if you're not sure of anything - everyone has been a beginner at some point and it's never too late or too early to learn to code and C# opens many options to take skills to mobile, desktop, cloud or even for games and new devices such as HoloLens but the smallest and best application is an old concept known as a console application, this allows you to see all your code in once place and easily see the output - don't worry if you're not familiar with the console these days but many years ago it was the only option for an application but still be able to take advantage of the latest features.

To make things even easier you don't need to use any programming tools or IDE such as Visual Studio for C# but will instead use a website called dotnetfiddle.net. There you can type in your Code and see what happens when it runs and there's nothing extra to install. However you can of course install “Visual Studio 2019 Community Edition” on your PC, then “Create a new project” and select “Console App (.NET Core)” and then input any code into the “Program.cs” file as this is the same code from that point this workshop will cover if you want to go through the exercises using the full tools available, but to keep things simple at first it's recommended you use something like dotnetfiddle.net for now.

Output, Operators, Types & Variables and Input

Output

In this first example, you will learn about how to write some code to display **Hello World**, this is a classic programming example that's traditional start to learning any programming language!

Visit **dotnetfiddle.net** and in the large box define which **namespace** to use entering the following:

```
using System;
```

System is used to allow access to the **Console** to output **"Hello World"**, a **namespace** is a way of organising groups of programming elements that can be used in the programme and the using statement should end with a semicolon like most statements in **C#**.

The next thing to do is to define the **class** for the example called **Demo** by entering in to **dotnetfiddle.net** the following:

```
public class Demo
{

}
```

Public is a Keyword that defines the access level of a **class** and how it may be used by other parts of the code and in this case, anything can access it. The **class** Keyword is used to define classes and this also uses curly braces to define the scope of the **class**.

The next thing to define is the **Main** Method which is the entry point for the example and is the first code to be run - this Method should be defined as below:

```
public static void Main()
{

}
```

Like a **class**, a method can also have an access level and again this is **public** so anything can access it, **static** defines this Method as global but will go into this concept in more detail later and the **void** Keyword means the Method doesn't return a value and the brackets define any Parameters or none in this case, we'll learn about those later too.

Within the **Main** Method in **dotnetfiddle.net** the following should be entered:

```
Console.WriteLine("Hello World");
```

Console is a type in the **System namespace** that can be used, here the **WriteLine** method is called and is like the **Main** Method in the example as it is also declared as being **static**. Inside the brackets is a value being passed as a Parameter, this is a **string** which is a set of text characters, in this case **"Hello World"**, the whole application in **dotnetfiddle.net** should look as below:

```
using System;

public class Demo
{
      public static void Main()
      {
            Console.WriteLine("Hello World");
      }
}
```

You can then use the **Run** option to start the programme which will display the text "**Hello World"** in the lower large box on **dotnetfiddle.net** or if **Auto Run** is **Yes** then it will run automatically.

This programme structure will be used in all subsequent examples so they'll always start with the **using** statements, followed by defining the **class** then the **Main** Method the example will use as its entry point which is the first point that will be reached when the example is started.

Operators

C# supports many kinds of Operators which are symbols that specify which Operations such as mathematics. We'll cover some of the basic ones here such as addition and multiplication.

The first thing is to define the basic structure of a programme by entering the following in **dotnetfiddle.net** in the main window:

```
using System;

public class Demo
{
      public static void Main()
      {
            // Code
      }
}
```

// Code is a Comment which is defined by writing **//** before anything that can be included in the programme but won't be run as part of it such as reminders or explanations of what the code should do.

The first Operator to use is **+** which is used for addition so enter the following below **// Code** in **dotnetfiddle.net**:

```
Console.WriteLine(4 + 2);
```

Then select **Run** in **dotnetfiddle.net** and this will display the answer as the output, it is also possible to use other mathematical Operators so can change **+** to ***** for multiply, **-** for subtraction or **/** for division to see what happens such as entering the following in **dotnetfiddle.net**:

```
Console.WriteLine(4 * 2);
Console.WriteLine(4 - 2);
Console.WriteLine(4 / 2);
```

You should see the appropriate answers appear in the output for each Operator that has been used whether it is addition, multiplication, subtraction or division.

Types & Variables

C# is what's known as a strongly typed language where every Variable has a **type** as does every expression that has a value, there are many types available, some of which will be covered here.

The first thing is to define the basic structure of the code by entering the following in **dotnetfiddle.net** in the main window:

```
using System;

public class Demo
{
      public static void Main()
      {
            // Code
      }
}
```

The first way to use types is like algebra in mathematics, so enter the following below **// Code**:

```
int a = 5;
int b = 2;
Console.WriteLine(a + b);
```

int is the **type** which is an Integer which can contain large or small whole numbers, **a** is the name of the variable and **=** is used to assign the value to the type, you can try changing the operator to ***** for multiply, **-** for subtraction and finally **/** for division and check the answer it gives when use **Run** in **dotnetfiddle.net**

In mathematics, it of course it is possible to get answers that aren't whole numbers so will need to use a **type** that allows this so replace the code from before with the following:

```
double a = 5;
double b = 2;
Console.WriteLine(a / b);
```

double is a **type** that does support decimal places in numbers so need to remember to use the correct **type** and you can use **Run** to see the correct answer.

Variables aren't just used to do mathematics - they can be used to store anything for use in the application either one or many times and can be changed with new values.

You can try this with **string** which represents a series of characters by entering in to **dotnetfiddle.net** the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            string message = "Hello World";
            Console.WriteLine(message);
      }
}
```

Text and Numbers relate to things you encounter day to day and can relate to but there are also Types that are a bit more specific to programming, enter in to **dotnetfiddle.net** the following example:

```
using System;

public class Demo
{
      public static void Main()
      {
            bool a = true;
            bool b = true;
            Console.WriteLine(a && b);
      }
}
```

You can then run this in **dotnetfiddle.net**, **bool** is a **type** to store Boolean values which can either be **true** or **false**, in this example it will output **true** and uses another operator **&&** which means "and" so if both **a** and **b** are **true** the output will be **true**, it's also possible to use **||** which means "or" to produce the same output as if **a** or **b** are true, change the "**Console.WriteLine**" line to the following:

```
Console.WriteLine(!(a && b));
```

When run in **dotnetfiddle.net** this will output the opposite value as **!** means not so if something is **true** it becomes **false** and if it's **false** it becomes **true**.

Input

In the previous examples everything has been output to the **Console** but it is also possible to allow something to be Input from the **Console**.

To create a simple input example by using **dotnetfiddle.net** and entering the following:

```
using System;

public class Demo
{
	public static void Main()
	{
		string value = Console.ReadLine();
		Console.WriteLine(value);
	}
}
```

When Run in **dotnetfiddle.net** a "prompt" will appear below the code window **>** and can then type anything there and end input by typing enter or return, this will then be output.

Another example is to enter a number to be used in a calculation by entering in to **dotnetfiddle.net** the following:

```
using System;

public class Demo
{
	public static void Main()
	{
		int a = int.Parse(Console.ReadLine());
		int b = int.Parse(Console.ReadLine());
		Console.WriteLine(a + b);
	}
}
```

By typing in a value for **a** or **b** can then output the sum of these values, another feature shown is that types often have static Methods, in this case **Parse** which allows something to be converted to that **type** so when Run in **dotnetfiddle.net** you can enter **1** then **2** to get the answer **3**, it's possible to change the operator to ones used for **int** such as ***** for multiply, **-** for subtraction or **/** for divide – for that last one try changing **int** to **double** to avoid losing anything after a decimal point like when doing **5** divided by **2**.

Flow, Loops, Properties & Access and Methods

Flow

Just doing the same thing over and over doesn't a very interesting application, that's where decisions come in, a program can make a choice what to do based on a condition to choose what path it should follow in the programme and can be used for a variety of purposes such as checking values or validating any user input.

The **if** statement can be used for conditions, in **dotnetfiddle.net** enter the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            int number = 0;
            if (number == 0)
            {
                  number = 9 + 2;
            }
            Console.WriteLine(number);
      }
}
```

The **if** statement is followed by a conditional statement in brackets which checks if the number is equal to zero, if it is the action in the curly braces will occur, if not it won't, so try changing the **int** value of **number** in **dotnetfiddle.net** to the following:

```
int number = 1;
```

This will change the output value because the number is no longer equal to zero, you can use anything that worked for **bool** to control what happens within an **if** statement. It's also useful to follow the indentation used in the example as it makes it very easy to see if what flow the programme is going to perform so can avoid any problems or confusion later about what will happen inside the **if** statement, in that example only one path was used based on whether something was **true**.

When using Conditions, it may be that you want to do something when the value is **true** then do something different if it isn't, this is done with an **if** - **else** statement where **if** is combined with **else** so that either one or the other is performed based on the condition being **true** or **false**.

To use an **if**, **else** statement in **dotnetfiddle.net** enter the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            Console.WriteLine("Enter Number between 1 and 10");
            int number = int.Parse(Console.ReadLine());
            if (number >= 1 && number <= 10)
            {
                  Console.WriteLine(number + " is valid");
            }
            else
            {
                  Console.WriteLine(number + " is not valid");
            }
      }
}
```

if is followed by a conditional statement in brackets – which in this example checks if the number is greater or equal to one and is less than or equal to **10**, when this is true the programme will perform the actions in the first set of curly brackets, but when this is false or when it's something **else** it will perform the action after that in the second set of curly brackets of the **if** statement, when the program is run try entering different numbers that are between **1** and **10** or different as see the different conditions of the programme.

Loops

Another way you can do more things by repeating statements in a loop. The first type of loop is the **while** loop which is like the **if** statement but instead of performing its operation once it will perform it until the condition is satisfied.

To use a **while** loop in **dotnetfiddle.net** enter the following:

```
using System;

public class Demo
{
	public static void Main()
	{
		int number = 0;
		Console.WriteLine("Enter Number between 1 and 10");
		while(number < 1 || number > 10)
		{
			number = int.Parse(Console.ReadLine());
		}
		Console.WriteLine(number + " is valid");
	}
}
```

In this example, the **while** loop will continue until the number is lower than 1 or it is higher than 10, so if enter anything outside that range it will keep looping otherwise it will exit the loop, this is like **if** that the loop only happens as long as the condition is true.

It is possible to write a loop that checks the condition last with a **do while** loop, you can do this in **dotnetfiddle.net** by entering the following:

```
using System;

public class Demo
{
	public static void Main()
	{
		int number = 0;
		Console.WriteLine("Enter Number between 1 and 10");
		do
		{
			number = int.Parse(Console.ReadLine());
		} while(number < 1 || number > 10);
		Console.WriteLine(number + " is valid");
	}
}
```

Another kind of loop is the **for** loop which is ideal for use when you know how many times you want to loop or it's easy to work out how many times the loop needs to run.

You can write a **for** loop in **dotnetfiddle.net** and enter the following:

```
using System;

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

The **while** loop uses a single **bool** expression but the **for** loop uses three expressions, the first declares an **int** variable called number and sets it to **0**, the second is the conditional expression which will be **true** until the number is not less than or equal to **10**, the third expression is used to increment the value using another Operator **++** which adds **1** to the value, there's also **--** that subtracts **1** from a value.

Another type of **for** loop is a **foreach** which will loop through an Array or similar Object where the items to be looped through already exist – an array is just a list of values and is denoted by **[]** to use this and the **foreach** in **dotnetfiddle.net** enter the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            int[] numbers = { 1, 2, 3, 4, 5 };
            foreach(int number in numbers)
            {
                  Console.WriteLine(number);
            }
      }
}
```

Properties & Access

Programmes can already have Variables but it's also possible to have Properties which is a flexible way or reading, writing or computing the value of another field and allows data to be accessed easily. This other field may be set up as **private** which is an Access modifier which means it can only be set or read within the same class they are contained within.

If is then possible to create a **public** property which can be accessed from anywhere that uses this **private** field and do something different with that Variable, you can do this in **dotnetfiddle.net** by entering the following:

```
using System;

namespace Workshop
{
      public class Demo
      {
            private static double _seconds = 0;

            public static double Minutes
            {
                  get { return _seconds / 60; }
                  set { _seconds = value * 60; }
            }

            public static void Main()
            {
                  Minutes = 15;
                  Console.WriteLine("Minutes:" + Minutes);
                  Console.WriteLine("Seconds:" + _seconds);
            }
      }
}
```

In this example, the Variable or Member **_seconds** is declared as **private** it is also using the **static** keyword as the method it is being called from is also **static**, the Property **Minutes** is declared as public so can save the Minutes value into seconds using the Property.

When declaring **private** members for properties you can name them with an underscore in front so you know they'll be **private** and usually not used directly. When declaring a **public** Property these usually begin with a capital letter and when using multiple words each word should be capitalised – this is known as Camel Case.

Methods

In all the previous examples there has just been one **static** method of **Main** in the **Console** application and have even used methods such as **int.Parse**. A method is a function used in a class and contains a series of usually related statements in a programme, they can optionally accept Parameters which are values that can be passed in or return a value.

Methods are something that do something in the programme and should be named accordingly so it's clear what it does and it usually should only do one thing and allows you to avoid repeating yourself when writing a programme, to create a Method in **dotnetfiddle.net** enter the following:

```
using System;

public class Demo
{
      public static int Addition(int value1, int value2)
      {
            return value1 + value2;
      }

      public static int Subtract(int value1, int value2)
      {
            return value1 - value2;
      }

      public static void Main()
      {
            Console.WriteLine("Addition:" + Addition(9,2));
            Console.WriteLine("Subtract:" + Subtract(9,2));
      }
}
```

In this example **Addition** is declared as a **public** and **static** method and it takes two parameters – which are between the brackets and then returns the values added together and **Subtract** is similar but returns the values subtracted from each other.

Methods that can be written or declared can return a value but they don't have to return a value – these values use **void** in place of a return type, and they can either accept parameters or not and it is also possible to have methods that have parameters that are optional but present with a default value when none is passed in.

An example of void and optional parameters in **dotnetfiddle.net** by entering the following:

```
using System;
using System.Text;

public class Demo
{
      public static void Loop(string value, int loop = 1)
      {
            StringBuilder message = new StringBuilder();
            for(int i = 1; i <= loop; i++)
            {
                  message.AppendLine(value);
            }
            Console.WriteLine(message.ToString());
      }

      public static void Main()
      {
            Loop("Hello");
            Loop("Hello World", 10);
      }
}
```

In this example, there is a **public static** Method that doesn't return a value but it uses a **string** Parameter called value which must always be used, and a second **int** Parameter which is optional, this value is used to control a **for** loop inside the method. Another **namespace** is also used in the example for **System.Text** which contains the **StringBuilder** which is used here, an Instance or copy of this needs to be used - that's what **new** does. Then the **AppendLine** Method of it is used to add a new line to the **StringBuilder** and when the loop is complete this is then converted to a **string** using an Extension Method called "**ToString**", this is similar to how **int.Parse** was used before and the **Loop** Method is called with or without the loop value to show the different behaviour.

Exceptions, Events, Classes & Objects and Interfaces

Exceptions

Exceptions are errors caused by things that weren't supposed to happen, but it is possible to handle when things don't happen as expected in your programme.

This example will crash if incorrect input is entered, in **dotnetfiddle.net** enter the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            Console.WriteLine("Enter Number");
            int number = int.Parse(Console.ReadLine());
            Console.WriteLine(number);
      }
}
```

Type **Number** and an **exception** will occur **Input string was not in a correct format** this is Unhandled as there's nothing to cope with this happening and the programme crashes.

To handle an **exception** you need a **try** - **catch** block where the **try** is the code you might have a problem with and a **catch** to do something when this **exception** happens.

To use a "**try** - **catch**" in **dotnetfiddle.net** enter the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            try
            {
                  Console.WriteLine("Enter Number");
                  int number = int.Parse(Console.ReadLine());
                  Console.WriteLine(number);
            }
            catch(FormatException ex)
            {
                  Console.WriteLine("Input was not Valid");
            }
      }
}
```

In the example, there is a **try** block contains any code that might **throw** an Exception, if this does happen then the **catch** block code is run, there is a parameter for **FormatException** which is the type of **exception** expected if the input is invalid, it's possible to get details of the error from the parameter if needed.

It's also possible to **throw** your own exceptions if something occurs and your programme shouldn't handle that situation itself but make the programme aware of this error instead.

To use a **throw** in **dotnetfiddle.net** by entering the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            Console.WriteLine("Enter Number between 1 and 10");
            int number = int.Parse(Console.ReadLine());
            if(number < 1 || number > 10)
            {
                  throw new ArgumentOutOfRangeException(
                  "Number must be between 1 and 10");
            }
      }
}
```

When you throw an exception using the **throw** Keyword followed by the type of exception object in this case it is an **ArgumentOutOfRangeException** to indicate when the input was out of range, this example would also raise the **System.FormatException** as well.

Something to remember is that it is best to not **throw** a **System.Exception**, **System.SystemException** or **ApplicationException** but more specific ones like the one in the example and that you shouldn't rely on Exceptions to do validation but use **if** to see if something is correct or not and use **exception** for those things that might happen such as you accept an **int** but the value entered is too high for that type.

Events

Events are a way for a **class** to provide notifications when something interesting has happened to an object, for example in a user experience with a button when you tap the button this would be an event that has occurred, however they aren't just for user experiences but are useful to indicate a change of state that might be of use to something.

Events in can be created using something known as a **delegate** which is a **type** that represents or encapsulates a Method with a set of parameters and a return **type**.

An example of using an **event** and **delegate** in **dotnetfiddle.net**, enter the following:

```
using System;

public class Demo
{
      public delegate void ChangedHandler(int number);

      public static event ChangedHandler Changed;

      private static void OnChanged(int number)
      {
            Console.WriteLine(number);
      }

      public static void Main()
      {
            Changed += new ChangedHandler(OnChanged);
            Changed(10);
      }
}
```

In this example, there is the **delegate** which is has the Signature of the **event** to use that would be the parameters it will use in this case it is an **int** called **number**. Then there is the **event** which uses the Delegate as its **type**, this would be what would be called to make the **event** occur. Then there is a Method which also matches the Signature of the **delegate** and takes the same Parameters. In the **Main** Method, there is the **event** and a new operator **+=** which in this case is used specify the method that will be called in response to an event which is a method with the same signature as the **delegate** and the **OnChanged** Method to be called when the event occurs, then we raise the **event** by calling that Method and pass in a value.

Classes & Objects

C# uses a lot of types – in fact it's known as a Strongly-typed language so that **int** can't be set to anything other than a whole number or **double** can be whole and decimal numbers and neither of those can contain a **string**.

A **class** is the definition or design of an **object** – an object is an Instance of a **class** which when you've been using **int**, **string** or **bool** those have been objects of those types, in **dotnetfiddle.net** you've been using one **class** called **Demo**, like the following:

```
public class Demo
{

}
```

You can create your own **class** to contain the Methods, Properties and Members" that are needed to represent a particular thing or group certain functionality together and all Classes derive from **object** so you can make your own **type** and have it work how you want, you can use a **class** to represent something from the real-world too.

When using multiple Classes in a programme you need a way to group them all together much like you put methods and properties into a **class** to organise Classes too, the way of doing this is by creating a **namespace**, they also use curly braces to define what goes inside the **namespace**, like a class does to define its scope, so at the top of your application you use **using** to import any **namespace** such as **System** then below this you define your own **namespace** to contain all the "class" items you create, here's a simple example showing this new structure:

```
using System;

namespace Workshop
{
      public class MyClass
      {

      }

      public class Demo
      {

      }
}
```

To write your own **class** to represent something in **dotnetfiddle.net** enter the following:

```
using System;

namespace Workshop
{
      public class Person
      {
            public string Forename { get; set; }
            public string Surname { get; set; }
      }

      public class Demo
      {
            public static void Main()
            {
                  Person item = new Person()
                  {
                        Forename = "John",
                        Surname = "Smith"
                  };
                  Console.WriteLine(
                  item.Forename + ' ' + item.Surname);
            }
      }
}
```

The **class** is used to represent a person and is called as so and appeared as the following:

```
public class Person
{
      public string Forename { get; set; }
      public string Surname { get; set; }
}
```

It is declared as **public** so it can be used without restriction and it has two Properties these also use a different way of writing the **get** and **set** so they just are **string** properties without a **private** member to represent them, but a value such as **_forename** still could be used, the class used appeared as the following:

```
Person item = new Person()
{
      Forename = "John",
      Surname = "Smith"
};
```

In the **class** of **Demo** an Instance of the **class** is created, this is done by using the name of the **class**, in this case **Person** which takes the place of the **type**, then is followed by a name and this is set to a **new** item of the **class** which is how the Instance is created, then within the curly braces the Properties of the **class** are set to some values.

Composition allows the creation of a **class** composed of other objects to enable more flexibility and information that can be stored in an Instance of a class, for example could expand the **class** of **Person** as for example as the following:

```
public class Contact
{
      public string Email { get; set; }
      public string Phone { get; set; }
}

public class Person
{
      public string Forename { get; set; }
      public string Surname { get; set; }
      public Contact Contact { get; set; }
}
```

In this example, the **class** of **Contact** has been added as another **class** to represent **Contact** details such as **Email** and **Phone**, see if can figure out how to set and output the **Email** Property.

Classes can share Properties and functionality of a Base **class** can use inheritance to Inherit or share functions of another **class**, this allows you to create objects that have common features, there's many real-world examples where physical objects share similar features and in programming you can do much the same thing and allows you to avoid repeating yourself, the language and it's features are designed to help you make sure you only do something once, or reuse something many times where needed.

An example of how to use inheritance in a programme by entering the following example into **dotnetfiddle.net**.

```
using System;

namespace Workshop
{
      public class Shape
      {
            public int Height { get; set; }
            public virtual double Area()
            {
                  return 0;
            }
      }

      public class Rectangle : Shape
      {
            public int Width { get; set; }
            public override double Area()
            {
                  return (Height * Width);
            }
      }

      public class Triangle : Shape
      {
            public int Base { get; set; }
            public override double Area()
            {
                  return 0.5 * Base * Height;
            }
      }

      public class Demo
      {
            public static void Main()
            {
                  Rectangle rectangle = new Rectangle()
                  {
                        Height = 10,
                        Width = 15
                  };
                  Triangle triangle = new Triangle()
                  {
                        Height = 10,
                        Base = 15
                  };
                  Console.WriteLine(rectangle.Area());
                  Console.WriteLine(triangle.Area());
            }
      }
}
```

Inheritance can use a Base **class** to define any Properties or Methods that will be common to all the Classes that will Inherit or derive from the **class**, shown in the following example:

```
public class Shape
{
      public int Height { get; set; }
      public virtual double Area()
      {
            return 0;
      }
}
```

In this example is a Property which will be a common Property for any **class** that will inherit this, also there is a Method which has been marked as **virtual** which allows any child Classes. Classes that use this class as their Base **class**, to Implement their own behaviour for this Method such as in the following example:

```
public class Rectangle : Shape
{
      public int Width { get; set; }
      public override double Area()
      {
            return (Height * Width);
      }
}
```

This **class** inherits from the **Shape class** and adds its own unique Property and implements its behaviour of that **virtual** Method from the Base **class** using the **override** Keyword, both the **Rectangle** and **Triangle** Classes both implement their own way of working out the **Area** but the result and Method is used the same way by anything that uses either child **class**, you can compare **Rectangle** with other the **class** of **Triangle** in the following example:

```
public class Triangle : Shape
{
      public int Base { get; set; }
      public override double Area()
      {
            return 0.5 * Base * Height;
      }
}
```

Where the **Area** is worked out for a **Triangle** differently that the **Rectangle**, demonstrating the flexibility and usefulness of inheritance.

Interfaces

An **interface** is defined in a similar way to a **class** but there are no keywords such as **public** and **private** nor are the statement blocks for a Method present and their name should start with **I** for Interface, Classes can implement many Interfaces, but only Inherit from a one **class**.

To use an "interface" in **dotnetfiddle.net** enter the following:

```
using System;
namespace Workshop
{
      public interface IShape
      {
            double Area();
      }

      public class Shape
      {
            public int Height { get; set; }
            public virtual double Area()
            {
                  return 0;
            }
      }

      public class Rectangle : Shape
      {
            public int Width { get; set; }
            public override double Area()
            {
                  return (Height * Width);
            }
      }

      public class Demo
      {
            public static void Main()
            {
                  Rectangle rectangle = new Rectangle()
                  {
                        Height = 10,
                        Width = 15
                  };
                  Console.WriteLine(rectangle.Area());
            }
      }
}
```

Libraries, Collections, Lambdas & LINQ and Generics

Libraries

Throughout the examples used with **dotnetfiddle.net** the **using** has been put at the top to include a **namespace**, this is taking advantage of the **.NET Framework** which features many kinds of "**namespace**" that can be used in an application that add features both complex and simple so many things don't need to be done again if they've been done already and many of these are contained in the **.NET Framework class** Libraries such as **System** and **System.Text** used in previous examples.

To use another **namespace** in **dotnetfiddle.net** enter the following:

```
using System;
using System.Text;
using System.Xml;

public class Demo
{
      public class Person
      {
            public string Forename { get; set; }
            public string Surname { get; set; }
      }

      public static void Main()
      {
            Person item = new Person()
            {
                  Forename = "John",
                  Surname = "Smith"
            };
            StringBuilder output = new StringBuilder();
            using(XmlWriter writer = XmlWriter.Create(output))
            {
                  writer.WriteStartElement("person");
                  writer.WriteStartAttribute("forename");
                  writer.WriteValue(item.Forename);
                  writer.WriteEndAttribute();
                  writer.WriteStartAttribute("surname");
                  writer.WriteValue(item.Surname);
                  writer.WriteEndAttribute();
                  writer.WriteEndElement();
            }
            Console.WriteLine(output.ToString());
      }
}
```

Namespaces used in the previous examples were **System**, **System.Text** and **System.Xml** – the first is where the main and common features such as the **Console** Methods that have been used like **Console.WriteLine**. **System.Text**" is used for **StringBuilder** which has also been used before, in this case it's used in combination with **System.Xml** to create an **XmlWriter** – this is used to create an XML document which is a way of storing data that can be read easily by a program or in the case written. There's another use of **using** here to create the **XmlWriter** to make sure it's created at the top part and then closed or Disposed when finished.

Another example is to use **XmlReader** to read in XML and take advantage of another **namespace** which is **System.IO** which allows input and output methods to be used in this case a **StringReader** to read some existing XML as an input for an application.

In **dotnetfiddle.net** enter the following example:

```
using System;
using System.IO;
using System.Xml;

public class Demo
{
      public class Person
      {
            public string Forename { get; set; }
            public string Surname { get; set; }
      }

      public static void Main()
      {
            Person item = new Person();

            string input =
            "<person forename='John' surname='Smith'/>";

            using(XmlReader reader = XmlReader.Create(
            new StringReader(input)))
            {
                  reader.ReadToFollowing("person");
                  item.Forename = reader.GetAttribute("forename");
                  item.Surname = reader.GetAttribute("surname");
            }
            Console.WriteLine("Forename:" + item.Forename);
            Console.WriteLine("Surname:" + item.Surname);
      }
}
```

Collections

It's possible to store multiple values of the same type, mentioned previously in the form of an Array, these are normally declared with a fixed size and denoted using square brackets and can be used to store a sequence of values of that **type**.

To use an Array in **dotnetfiddle.net** enter the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            string[] letters = { "A", "B", "C", "D", "E", "F" };
            foreach(string letter in letters)
            {
                  Console.WriteLine(letter);
            }
      }
}
```

It is also possible to loop through an Array by Index where the first element is zero.

To loop through an Array this way, enter **dotnetfiddle.net** the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            string[] letters = { "A", "B", "C", "D", "E", "F" };
            for(int index = 0; index < letters.Length; index++)
            {
                  Console.WriteLine(letters[index]);
            }
      }
}
```

Arrays are built into **C#** and can be utilised for many things but they have limitations that their size must be known in advance and often you won't know what to store beforehand, there is another way to store multiple items is with a collection.

There's another way of storing items of a **type**, this is known as a Collection and there are two main kinds which are **List** and **Dictionary**.

The first type of collection is **List** where it is a list of a particular item so if it's a **List** of **int** this would be **List<int>** where the **type** of the list is within angle brackets, they are more flexible than an Array as it can have as many items as you want and Collections have their own **namespace** which is **System.Collections.Generic**.

To use a "List" of "int" in **dotnetfiddle.net**, enter the following:

```
using System;
using System.Collections.Generic;

public class Demo
{
	public static void Main()
	{
		List<int> numbers = new List<int>();
		for(int index = 1; index <= 10; index++)
		{
			numbers.Add(index);
		}
		foreach(int number in numbers)
		{
			Console.WriteLine(number);
		}
	}
}
```

It's also possible to use a **List** of **string** in **dotnetfiddle.net**, enter the following:

```
using System;
using System.Collections.Generic;

public class Demo
{
	public static void Main()
	{
		List<string> letters = new List<string>()
		{ "A", "B", "C", "D", "E", "F" };
		foreach(string letter in letters)
		{
			Console.WriteLine(letter);
		}
	}
}
```

In the first example, there is a **List<int>** of numbers which is added to using the **Add** Method to add something to the **List** within a **for** loop. Then in the second **foreach** loop the items that were added are output. Then in the second example there is a **List<string>** of letters which have been prepopulated, like the Array and then the contents are output from a **foreach** loop.

The second type of collection is **Dictionary** which is like **List** but has two parts, there is the **Key** which will identify something that has been added, and **Value** which is the item that's been added and like **List** the **Dictionary** can have many types of **Value** but can also have types of **Key** but mainly **string** is the most commonly used **type** for a **Key** and you can get values of a **type** by their **Key**.

To use a "**Dictionary**" by **string** of **string** in **dotnetfiddle.net**, enter the following:

```
using System;
using System.Collections.Generic;

public class Demo
{
      public static void Main()
      {
            Dictionary<string, string> colours =
                  new Dictionary<string, string>();
            colours.Add("R", "Red");
            colours.Add("G", "Green");
            colours.Add("B", "Blue");
            foreach(string key in colours.Keys)
            {
                  Console.WriteLine(colours[key]);
            }
      }
}
```

In the example, there's a **Dictionary** of **string** where the **Key** is also a **string**, it has an **Add** Method with the **Key** and then the **Value** there there's loop which uses the **Keys** Property which is a **List** of the **Keys** and then output the item using the **Key** similar to an Array by using the square brackets.

Lambdas & LINQ

C# has a special type of Method that can be used as an alternative to creating another named Method you use but are created in-line with your code, they can be passed as a Parameter to other methods and these are known as Lambda expressions.

To use a Lambda expression in **C#**, in **dotnetfiddle.net** enter the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            Func<int, int> addTen = x => x + 10;
            Console.WriteLine(addTen(4));
      }
}
```

In the example, there is a **Func<int, int>** which is a variable the Lambda is assigned to then to the right of the **=** is the Lambda expression and there is two parts, the first is the Parameter passed in and then there is a special operator **=>** pronounced as "produces" which is followed by the code to execute and can be done in one line and doesn't require a **return** statement like a Method that returns a value would normally need.

To use multiple Parameters with a Lambda in **dotnetfiddle.net** by entering the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            Func<int, int, int> area = (x,y) => x * y;
            Console.WriteLine(area(4,5));
      }
}
```

Lambda expressions are a powerful and easy way to access functionality without needing a separate Method and can be reused in many places just like a Method but also passed into a Method like a Value can be.

The language of **C#** is powerful but to add to this power is a way of querying sets and lists of data using **LINQ** or **Language-Integrated Query**, which is a set of Extension Methods which are a special kind of **static** Method to extend existing types with new functionality and in **LINQ** this is used extensively and makes performing what would be complex tasks much easier and just requires the inclusion of another **using** for **System.Linq** and they also make use of Lambda expressions.

The following examples will add to the previous, so to start will need a **class** and then a collection of them as a **List** by entering in to **dotnetfiddle.net** the following:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Workshop
{
      public class Person
      {
            public string Forename { get; set; }
            public string Surname { get; set; }
            public int Age { get; set; }
      }

      public class Demo
      {
            public static void Main()
            {
                  List<Person> people = new List<Person>();
                  people.Add(new Person { Forename = "John",
                  Surname = "Smith", Age = 30 });
                  people.Add(new Person { Forename = "Jane",
                  Surname = "Smith", Age = 42 });
                  people.Add(new Person { Forename = "John",
                  Surname = "Doe", Age = 24 });
                  people.Add(new Person { Forename = "Jane",
                  Surname = "Doe", Age = 58 });
            }
      }
}
```

In this example, there is a **class** of **type** of **Person** which has two Properties of **string** and another of **int** then there is a **List** of **Person** which is then added to or populated with some values and this example will be added to in subsequent examples.

LINQ allows you to search for items within a collection that match a set of criteria and is the main component of **Language-integrated Query**, which is to Query data, which can be done with **where**.

To use "**where**" with **LINQ** in **dotnetfiddle.net**, add into **Main** at the end the following:

```
IEnumerable<Person> overFourty = people.Where(x => x.Age > 40);
Console.WriteLine(overFourty.Count());
```

In this part of the example there is the use of **IEnumerable<Person>** which is the type of item that. The **where** uses a Lambda and the expression looks for any item that is in the **List** which has an **Age** value greater than **40** and then outputs this value which should be **2** using another **LINQ** method of **count**.

You may have too many items in the collection or not have enough properties and you can use the **select** Method in **LINQ** which can be a useful way of converting an existing Collection into other collections or values.

To use **select** with **LINQ** in **dotnetfiddle.net**, add below last example the following:

```
IEnumerable<string> foreNames = people.Select(x => x.Forename);
foreach(string foreName in foreNames)
{
      Console.WriteLine(foreName);
}
```

In this part of the example the **select** is used to get the Forename for each **Person** in the Collection and will return a collection of **string** Objects that are then looped though with a **foreach** statement to output each Forename.

You may just want to get one item from a collection and you can do this in **LINQ** with **First**, **Last** and **Single** to get an item or **FirstOrDefault**, **LastOrDefault** and **SingleOrDefault** where **OrDefault** is the default value for a **type** which is usually **null** in case the value being looked for in the collection is not present.

To use **first** & **last** with **LINQ** in **dotnetfiddle.net**, add below last example the following:

```
Person first = people.First();
Person last = people.Last();
Console.WriteLine(first.Surname);
Console.WriteLine(last.Surname);
```

It is also possible to get information about a collection using **LINQ** with **count**, **any** and **all** - **count** was used previously and will return the number of items in the collection but it can also take a Lambda" expression or Predicate like a **where** does so you could count only certain things that match the expression. With **any** this will check the collection and see if anything that matches the Predicate expression matches and will return **true** if it does, and **false** if not and **all** which is like **any** except that all values must match to return **true** otherwise it will return **false**.

To use **count** in **dotnetfiddle.net**, add below the last example the following:

```
Console.WriteLine(people.Count(x => x.Age < 50));
```

To use **any** in **dotnetfiddle.net**, add below the last example the following:

```
Console.WriteLine(people.Any(x => x.Surname == "Doe"));
```

To use **all** in **dotnetfiddle.net**, add below the last example the following:

```
Console.WriteLine(people.All(x => x.Forename == "John"));
```

The results of many of the **LINQ** queries shown have been of **IEnumerable** of a type such as **int** or **string**, but there may be times where you need a different **type** than **IEnumerable** and **LINQ** has Extension Methods that allow you to do this.

To get a **List** of the type returned by **LINQ** in **dotnetfiddle.net**, enter the following:

```
List<string> list = people.Select(x => x.Forename).ToList();
```

To get an Array of the type returned by **LINQ** in **dotnetfiddle.net**, enter the following:

```
string[] array = people.Select(x => x.Forename).ToArray();
```

LINQ is a very powerful addition to **C#** and there are many more ways you can use it to get various parts of a collection, think of it as a search engine for your code, whatever information is in there, you can get at it with **LINQ** expressions and can use them in combination with each other to make your applications much more powerful than they would otherwise be and makes it easy and straightforward to manipulate data.

Generics

C# contains many powerful features that are useful to allow the creation of software that can reuse or simplify many complex but common programming tasks, as projects become more complex there needs to be a better way to reuse code, to help with this **C#** includes a feature called Generics. Generics allows a **class** to take type parameters allow for more functionality in much the same way as Methods can be made more powerful because they take parameters. Generic Classes and Methods combine reusability and efficiency in a way that non-generic alternatives can't, they are most frequently used with collections in fact **List** and **Dictionary** used previously are an example of Generics. When you encounter the **<** and **>** such as in **List<int>** the **type** within them can be any type which is what makes them Generic – that is they can be any **type** of **List** or **Dictionary** this type is usually indicated by using **T** and referred to as being **type** of **T** where **T** is the **type**.

To use Generics in **C#** enter in to **dotnetfiddle.net** the following:

```
using System;

namespace Workshop
{
      public class Container<T>
      {
            private T _value;

            public Container(T val)
            {
                  this._value = val;
            }

            public T Get()
            {
                  return this._value;
            }
      }

      public class Demo
      {
            public static void Main()
            {
                  Container<string> name =
                        new Container<string>("John Smith");
                  Console.WriteLine(name.Get());
            }
      }
}
```

In the example, there is a Container **class** which takes **type** of **T** denoted by **<T>** which will be the **type**, then a value which is of **type** of **T** and a **Get** Method which returns **type** of **T**. Although this is a simple example the Container **class** could be used with any **type**.

To use another **type** in the previous example below the **Console.WriteLine(name.Get());** line in **dotnetfiddle.net** enter the following:

```
Container<int> a = new Container<int>(4);
Container<int> b = new Container<int>(5);
Console.WriteLine(a.Get() + b.Get());
```

You can also use Generics to combine functionality needed such as creating a **new** instance of a **class** whatever its type might be using with a Factory **class** with a Create Method to create a new instance of the **class**.

To implement a basic Factory class in **C#**, enter in to **dotnetfiddle.net** the following:

```
using System;

namespace Workshop
{
      public class Factory<T> where T : new()
      {
            public T Create()
            {
                  return new T();
            }
      }

      public class Person
      {
            public string Name { get; set; }
      }

      public class Demo
      {
            static Factory<Person> factory =
            new Factory<Person>();

            public static void Main()
            {
                  Person person = factory.Create();
                  person.Name = "John Smith";
                  Console.WriteLine(person.Name);
            }
      }
}
```