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.