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());
		}
	}
}
```