Visual C# working with objects and classes

C# is an object orientated language. Things are defined as a class, this is a set of properties and behaviours. We often work with an instance of a class known as an object.

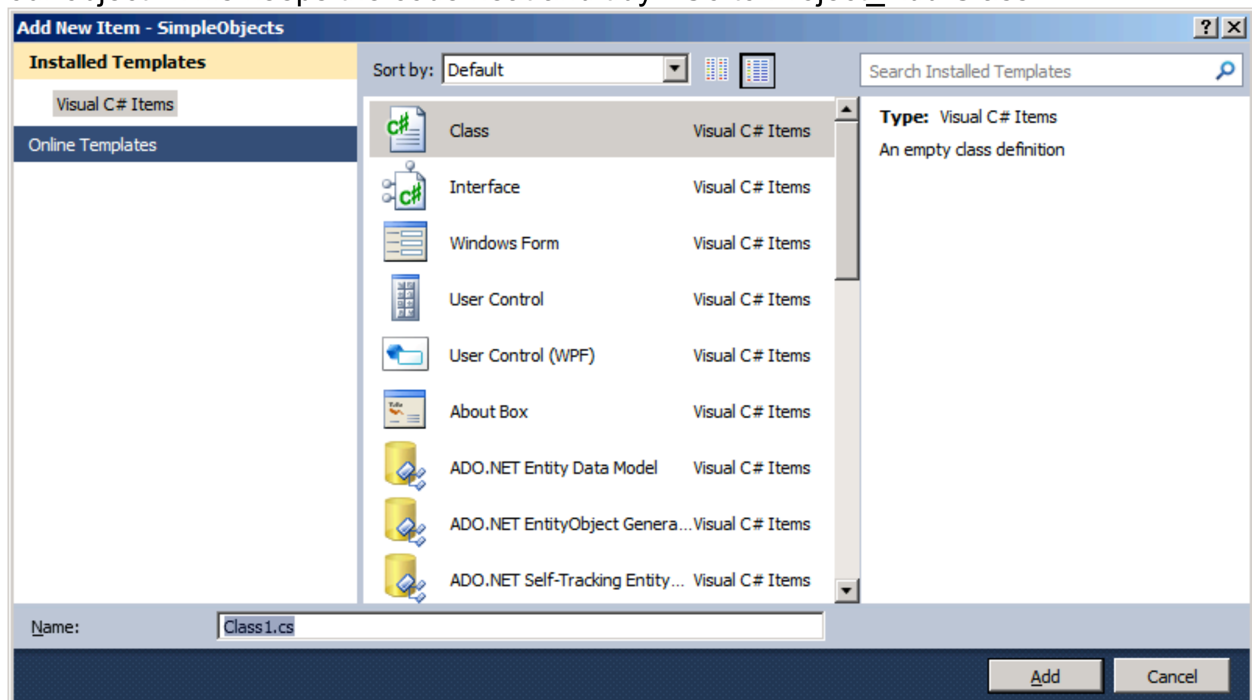
The Form is a class but we create a separate object from each form to use in our projects. The following line states that inherits `frmSeeMonster` all the behaviours of a Form (square, grey, closing and resizing icons) but we can add additional behaviours of our own (Buttons, Labels, code).

```
public partial class frmSeeMonster : Form
```

All the visual widgets (Buttons, Labels) are classes as are many other C# components that can be used but not seen. Int16 (used to parse from a String to an int) is a class. A String is a class, an array (int [] myArray) is not a class but Array is.

The simple rules is that a class begins with a capital letter but an object (an instance of a class) begins with a small letter.

To begin we will create a Form based project and add an additional class to define our object. This keeps the code neat and tidy. Go to Project_Add Class



Give the class a sensible name, here our class name will be Monstrer.cs.

This is the complete class code. The class must have the same name as the file.

```
class Monster
{
    //this is a private property of Monster
    private String mName;
    public Monster()
    { //this is the constructor, it is called when the
        // object is instantiated
        mName = "unknown";
    }
    public String getName()
    { //this public method will return the name of the monster
        return mName;
    }

}
```

We need a means to manipulate the class so will create an instance of the class within the visual Form. For testing a single Button (bTest) and Label (lblTest) are on the form. Clicking the Button will instantiate the Monster object.

```
Monster myMonster = new Monster();
lblTest.Text = myMonster.getName();
```

Note that we work with the object myMonster not the class Monster.

The Monster so far is pretty limited. All it has is a name and we cannot change that.

First give it some more attributes. It is good practice to set these with default values in the constructor.

```
private String mName;
private int hitPoints;
private int strength;
public Monster()
{
       mName = "unknown";
       hitPoints = 10;
       strength = 10;
}
```

To see these new attributes additional getter methods will need to be defined.

```
public int getStrength()
{
       return strength;
```

```
}
```

These can then be picked up by the object myMonster.

It is useful to be able to change the attributes of the object. This is achieved by setter methods. These return void but accept a variable to assign to the attribute of the object.

```
public void setName(String theName)
{
       mName = theName;
}
```

Another key development is that we can have several constructors. These can be used to assign values to attributes other than the default when the class is first instantiated. Note that a constructor does not return a value.

```
public Monster()
{
     mName = "unknown";
     hitPoints = 10;
     strength = 10;
}
public Monster(String theName)
{
      mName = theName;
      hitPoints = 10;
      strength = 10;
}
```

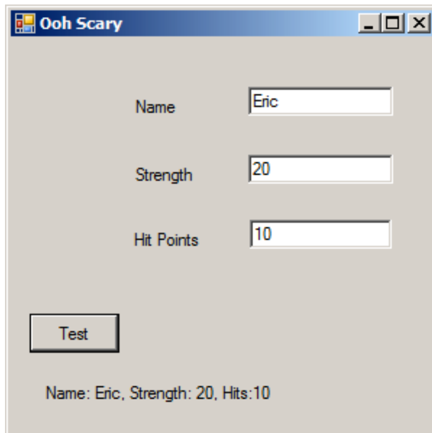
This class could have 2 more constructors to set the hitPoints and strength at instantiation.

Here the data for name, hitPoints and strength is picked up from TextBoxes and output back to a Label.

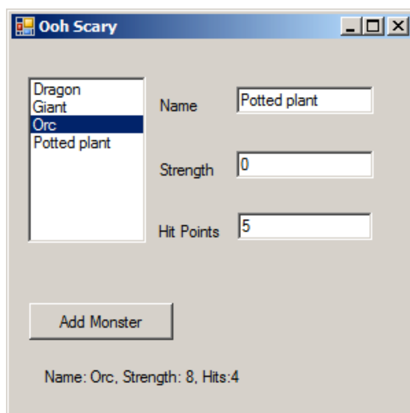
This is the code required by the Form front end.

```
Monster myMonster = new Monster(txtName.Text,
Int16.Parse(txtHits.Text), Int16.Parse(txtStrength.Text));

lblTest.Text = "Name: " + myMonster.getName() + ", Strength: " +
myMonster.getStrength().ToString()
                + ", Hits:" + myMonster.getHitPoints().ToString();
```



This is all fine but we are only using 1 monster. The power of classes comes when we have several objects each with the properties of the class.



Here the ListBox will display all the Monsters that have been created with the Button. Double clicking on each list item will show the monster statistics. The Label is displaying the selected monster but the TextBoxes show the statistics of the last monster entered (the not so deadly potted plant).

We can directly put a Monster object into a ListBox. To make the user interface neater and allow for other processing the Monster objects are placed in an array and only the Monster name is shown by the ListBox. The position of the Monster name in the ListBox is also the position of the Monster object in the array.

Declare the array and a counter at the top of the Form.

```
Monster[] monsterList;
int monsterCount = 0;
```

The array is instantiated in the Form constrructor. A length of 16 has been chosen; this could be changed later.

```
monsterList = new Monster[16];
```

The Button creates a Monster then adds it to the array and its name to the ListBox.

```
monsterList[monsterCount] = myMonster;
```

```
monsterCount++;
lstMonster.Items.Add(myMonster.getName());
```

In the double click event of the ListBox we get the Monster from the same selected index value.

```
Monster listMonster = monsterList[lstMonster.SelectedIndex];

lblTest.Text = "Name: " + listMonster.getName() + ", Strength: " +
listMonster.getStrength().ToString()
5 + ", Hits:" + listMonster.getHitPoints().ToString();
```

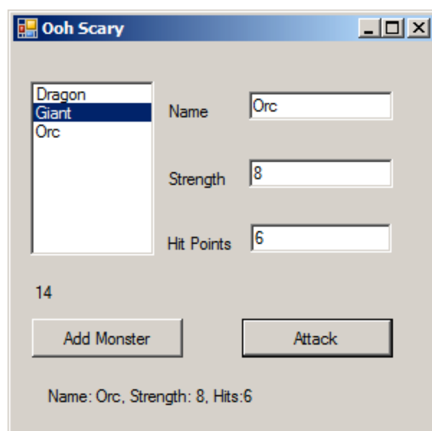
So far the classes have attributes but are acting as linked lists of values. The methods of a class are not restricted to getting and setting values. Decisions and action can be made.

Here a method of the class Monster simulates the Monster using its strength to attack.

```
public int attack()
{
        Random r = new Random();
        int damage = r.Next(0,strength);
        return damage;
}
```

This is used by a Monster object on the Form. Here a Button click will model the attack.

```
int pos = lstMonster.SelectedIndex;
if (pos != -1)
{
         Monster attacker = monsterList[pos];
         int dam = attacker.attack();
         lblDam.Text = dam.ToString();
}
```



Here the Giant not the Orc is attacking

Inheritance in C#

We could consider an adventurer as a special type of Monster. They can fight and take damage like a Monster. Additional functionality could include tracking treasure and experience. This is the code to ensure that the new class extends Monster with 2 new attributes for the Adventurer.

```
class Adventurer : Monster
{
        private int treasure;
        private int experience;
}
```

The sub class can access the methods of the superclass but to change its attributes these must be modified from private to protected.

```
class Monster
{
        //this is a private property of Monster
        protected String mName;
        protected int hitPoints;
        protected int strength;
}
```

The Adventurer class can now have its own constructors that use the attributes of Monster.

```
public Adventurer()
{
        mName = "Hero";
        hitPoints = 20;
        strength = 10;
        treasure = 0;
        experience = 0;
}
```

As much of the constructor code of the super and sub class will be much the same we can access the super or base class within the sub class constructor.

```
public Adventurer(String theName, int hits,
int theStrength):base(theName,hits,theStrength)
{
            treasure = 0;
            experience = 0;
}
```

The new class can now be instantiated as follows:

```
Adventurer myMonster = new Adventurer(txtName.Text,
Int16.Parse(txtHits.Text), Int16.Parse(txtStrength.Text));
```