OOPDraw:

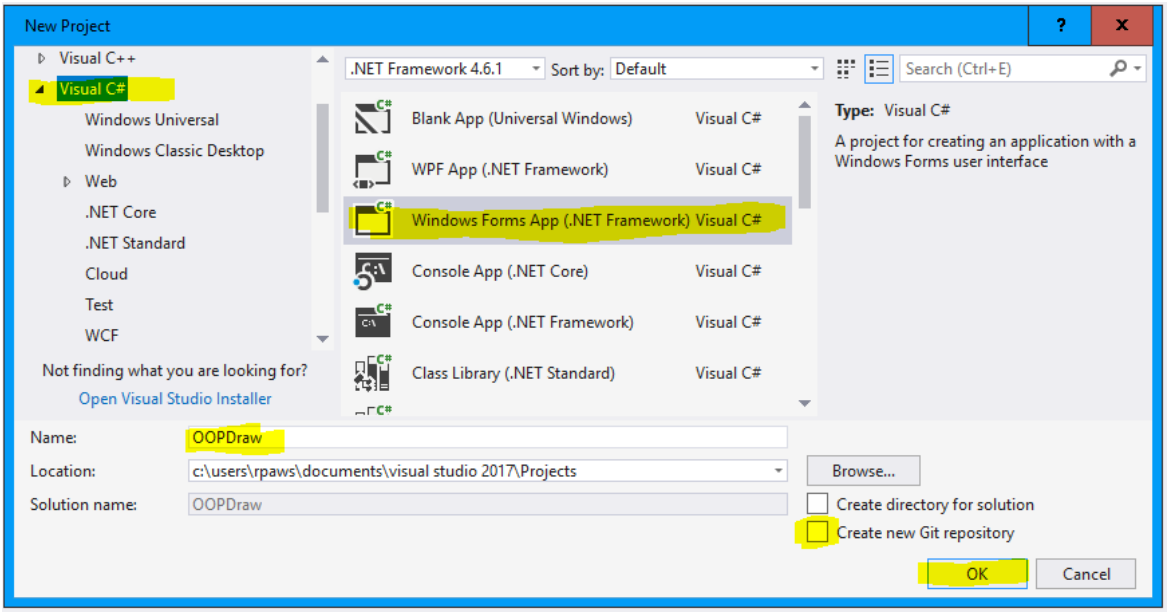
Learn the principles of OOP by writing a simple drawing program

STUDENT WORKBOOK

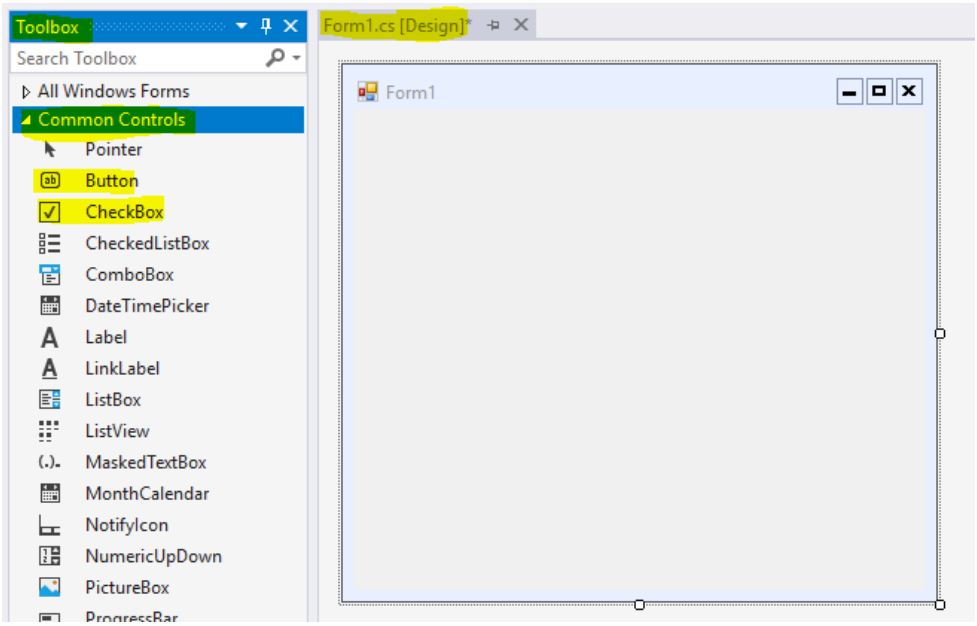
Exercise 1. Setting up the project

For this project we are going to create a Graphical User Interface (GUI) using Microsoft's Windows Forms (WinForms) technology.

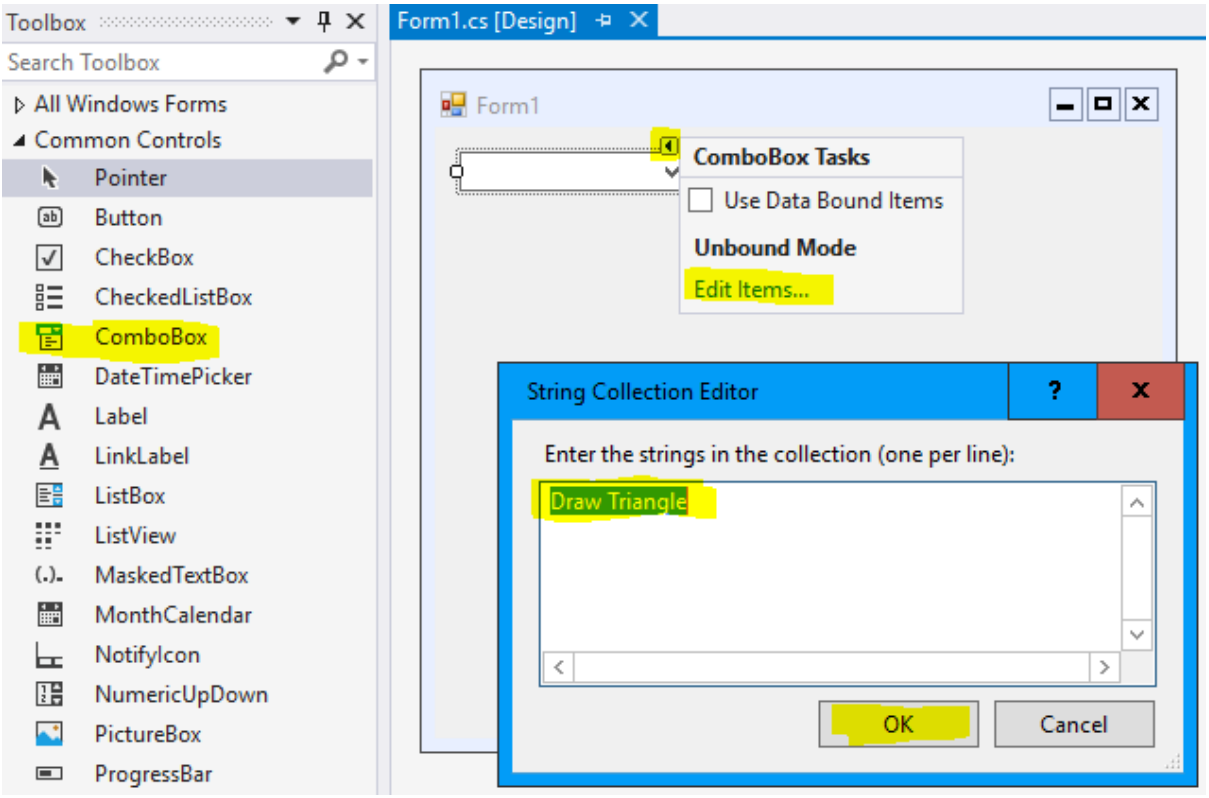
So start by creating a new project of this type, giving it a suitable name and saving it to a known location (if you are familiar with using Git you may choose to select the **Create new Git repository** option - otherwise leave it unselected):



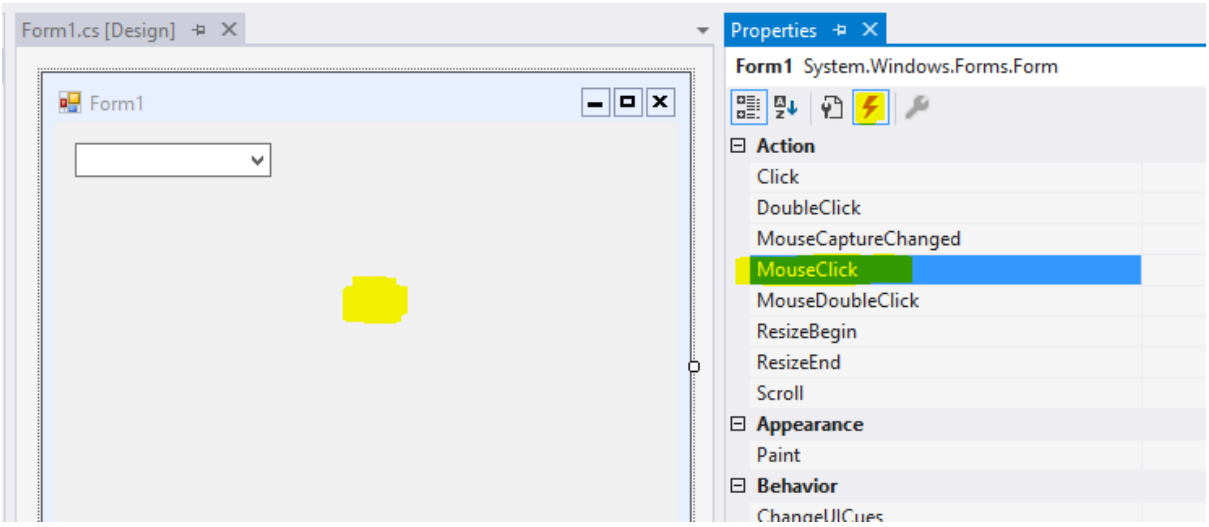
The project will be created along with the first Form, called **Form1**, which is all we will need for this project. It has been opened in 'Design' view - later we will be viewing and modify the (C#) 'code-behind' this Design view. You should also see the **ToolBox** with icons for adding various **Controls** such as **Button** and **CheckBox** to the form. (If you don't see the **ToolBox**, open it via the Visual Studio main menu: **View > ToolBox**).



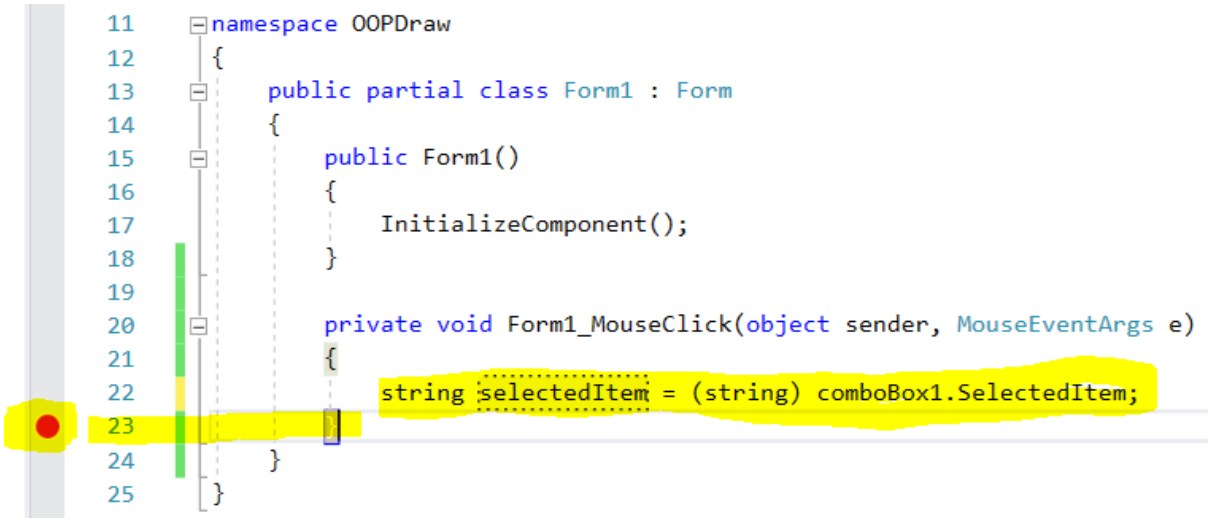
From the ToolBox drag a **ComboBox** control (to offer the user a drop-down list) into the Form, then click on the small triangle at the top of the element (highlighted in the screenshot below) and from the pop-up dialog, select **Edit Items**, and in the new dialog enter the text 'Draw Triangle' and hit **OK:**



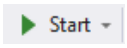
Now **right-click** anywhere within the form, *but not inside the ComboBox* control, and select Properties. (In the image below the properties pane has been moved alongside the Form pane for clarity.) Select the tab that shows 'events' - it has a lightning icon - and find **Action > MouseClick:**



Double-click on this **MouseClick** event and you will be taken to a view of the C# 'code behind' Form1, where a new function will have just been added. Add into this function in the new line of code highlighted below and then add a breakpoint on the line below it, by clicking where the red dot as shown:



```
11  namespace OOPDraw
12  {
13      public partial class Form1 : Form
14      {
15          public Form1()
16          {
17              InitializeComponent();
18          }
19
20          private void Form1_MouseClick(object sender, MouseEventArgs e)
21          {
22              string selectedItem = (string) comboBox1.SelectedItem;
23
24      }
25  }
```

If you now run the application in debug mode as usual (by hitting 

) you will be presented with the running form, which you may resize to full screen if you wish. Select the only option (**Draw Triangle**) in the ComboBox and then single-click somewhere else within the form. You will be taken to the breakpoint in the code, and if you hover on the variable `selectedItem`, you will see that it holds the text `Draw Triangle`. You can begin to see now how the code will interact with the GUI.

Note that `(string)` in the line above is needed to 'cast' the `SelectedItem` from `comboBox1` specifically to a string type.

Run the program again, but this time don't select anything from the ComboBox and just click within the form.

Q1 [Answer on EAD]	**What is the value of SelectedItem now?**

We're now ready to start some drawing, so stop the program before proceeding.

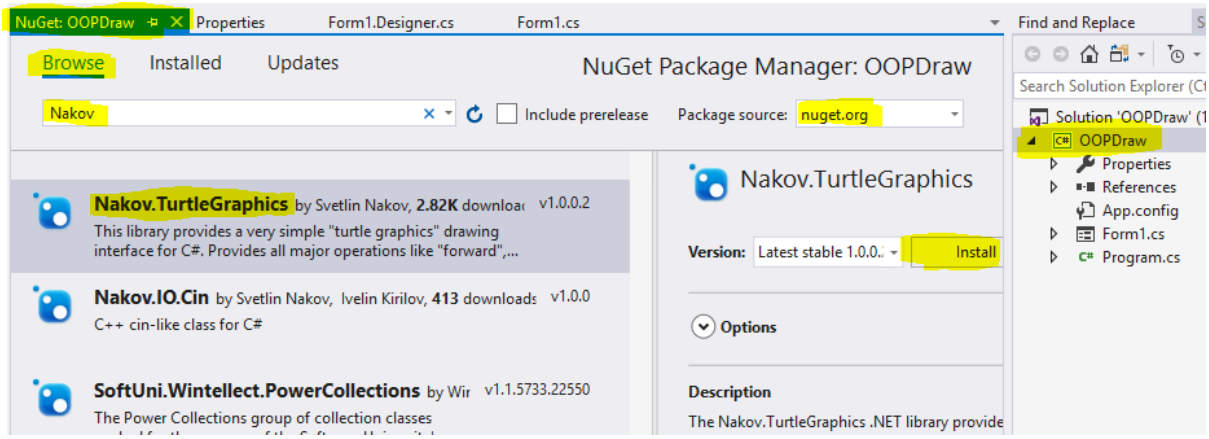
Exercise 2. Using Turtle Graphics

To create the lines that make up our drawing, we are going to use 'Turtle graphics'. You might well have encountered 'Turtle Graphics' earlier in your education - instructions are given to an imaginary Turtle to move forward/backward a specified distance, and to rotate (turn) a number of degrees (positive for clockwise, negative for anti-clockwise). You might well have learned this by simulating the turtle with your own body: taking 3 steps forward, turning right and so on.

Turtle graphics would not be a particularly efficient way to create a full-scale drawing program: you would more likely use a more powerful graphics library, one of which is built into .NET anyway. But there are several advantages to using Turtle graphics for this exercise:

- It offers a very simple way to draw line shapes
- It is inherently *procedural,* allowing us to focus on the more advanced *object-oriented* concepts that we want to introduce and build around the simple drawing procedures.
- More sophisticated graphics libraries would already have capabilities for drawing, moving and modifying shapes, leaving use little to learn.

We will use a free Turtle graphics library, called Nabov Turtle, and first we need to add this library to our code in the form of a 'package'. **Right-click** on the **OOPDraw** project icon, and select **Manage NuGet Packages**, to open the NuGet Package Manager shown below:



Browse the **nuget.org Package source** to find the **Nakov.TurtleGraphics** package (search just for 'Nakov') and then hit **Install.**

Now return to the Form1 code file. (If you've closed this pane, retrieve it by right-clicking on the Form1.cs item in the Solution Explorer and selecting **View Code)**, and add the code shown below. (You will also need to add `using Nakov.TurtleGraphics;` at the top of the file, though **Quick Actions** should give you a shortcut for this):

```
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    string selectedItem = (string)comboBox1.SelectedItem;
    if (selectedItem == "Draw Triangle") //We will add more options later
    {
        Turtle.Rotate(30);
        for (int i = 0; i < 3; i++)
        {
            Turtle.Forward(50);
            Turtle.Rotate(120);
        }
    }
}
```

Run the program select 'Draw Triangle' and click in the form.

Q2 [Answer on EAD] **Paste in a screenshot of the resulting triangle**

A few refinements are needed: get rid of that cartoon Turtle; thinner lines, perhaps. But let's also take the opportunity to move the code for drawing a triangle into a re-usable function, and one that allows us to specify the location and size (side length), and then call the function such that the triangle is drawn from the place on screen where the mouse is clicked:

```
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    //Transform windows coordinates to Turtle coordinates
    float turtleX = e.X - Width / 2 + 8;
    float turtleY = Height / 2 - e.Y - 19;
    string selectedItem = (string) comboBox1.SelectedItem;
    if (selectedItem == "Draw Triangle") //We will add more options later
    {
        DrawTriangle(turtleX, turtleY, 50);
    }
}
private static void DrawTriangle(float xOrigin, float yOrigin, float sideLength)
{
    Turtle.ShowTurtle = false;
    Turtle.PenSize = 2;
    Turtle.Angle = 0;  //Always start from North
    Turtle.X = xOrigin;
    Turtle.Y = yOrigin;
    Turtle.Rotate(30);
    for (int i = 0; i < 3; i++)
    {
        Turtle.Forward(sideLength);
        Turtle.Rotate(120);
    }
}
```

Note that the first two lines read the location of the mouse-click from the MouseEventArgs parameter, via `e.X` and `e.Y`. If you experiment you will discover that these values are relative to the top-left corner of the form. However, the Turtle coordinates have their origin in the centre of the form (and this moves if you change the size of the pane when running). So we need to perform a simple transformation on the coordinates - with a small offset at the end to allow for the border width.

Run the program again, to prove that the triangle is now drawing starting from where you click on screen. Moreover, if you click again in a different place, another triangle is drawn.

Q3 [Answer on EAD] **Paste in a screenshot showing several triangles drawn within the form.**

Now create a `DrawRectangle` function, similar to `DrawTriangle`. *Instead of* a single `sideLength` parameter, it should take `height` and `width` parameters (both of type `float`). Then modify the ComboBox to add the option 'Draw Rectangle' and connect-up your new function in the `Form1_MouseClick` function, calling it with values of 50, and 100 for height and width.

Q4 [Answer on EAD] **Paste in your code for the `DrawRectangle` function**

Q5 [Answer on EAD] **Paste in your code for the modified `Form1_MouseClick` function**

Q6 [Answer on EAD] **Paste in a screenshot showing that you have drawn a triangle and a rectangle in different places on the same screen.**

Exercise 3. Introducing objects as custom data types

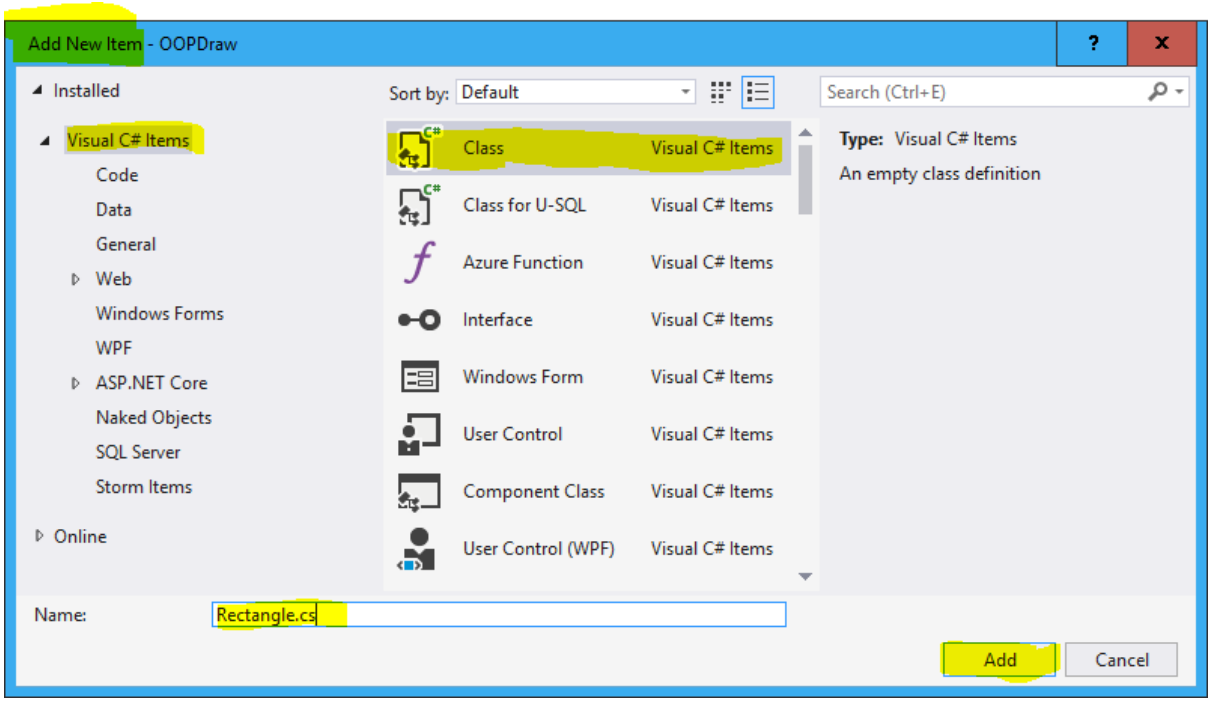
It would be possible continue with the *procedural* approach adopted so far to create a sophisticated drawing, and indeed this is somewhat like the approach used by *Paint* programs.

The limitation of this approach is that once a shape has been drawn, that shape exists only as a set of pixels on the screen. If we wanted to change the drawing (to change the size or position of one shape, say) we would have to erase the necessary pixels and draw something afresh. We would have to keep a record of each of the parameters for each of the shapes, modify the ones we want to change, and then call the shape-drawing functions again for each set of parameters.

At minimum it would be nice to hold all the parameters that define one rectangle, say, in one holder. So far, all the parameters have been of type `float`, so we could perhaps, hold the values in an array. But what happens when we want to add new parameter types such as Color, string, or Boolean? And how do we handle the fact the functions to draw a triangle and a rectangle already have a different numbers of parameters?

What we really want is to have our own types, in addition to float, string, integer (and so on), called, say, Rectangle and Triangle (to be more precise, EquilitateralTriangle in this case) which hold together all the values that define one rectangle or triangle.

An *object* can be thought of, in the first instance, as custom data type containing multiple *properties* holding individual data items, each with a name and type. (Shortly, we will see that this is a very limited view of what an object is, but it will do for the moment). We define one of these custom types by creating a *class.* You can define a new class within the same code file as your drawing functions, but it is considered better practice to create each new class in a file of its own. Do this by right-clicking on the project icon and selecting Add > Class, giving the new class the name Rectangle:



Then modify the created code as shown:

```
namespace OOPDraw
{
    public class Rectangle
    {
        //Properties
        public float XOrigin { get; private set; }
        public float YOrigin { get; private set; }
        public float Width { get; private set; }
        public float Height { get; private set; }

        //The 'Constructor'
        public Rectangle(float xOrigin, float yOrigin, float width, float height)
        {
            XOrigin = xOrigin;
            YOrigin = yOrigin;
            Width = width;
            Height = height;
        }
    }
}
```

Notes:

- We made the class `public` so that it can be used by code elsewhere in the program
- There are four properties, each holding a piece of information about a particular rectangle
- Code outside this class can read (`get` in OOP speak) each property but cannot modify (`set` in OOP speak) them because the `set` is marked `private`.
- A class can be thought of as a 'cookie cutter'. When the program is running *instances* (think of them as the actual cookies) are made with it. This is done by calling a *constructor* on the class. The constructor takes as parameters any values that you want to force the program to specify when they create any new instance. These are used in the body of the constructor to set the properties on the new instance. This might look like a lot of unnecessary effort and, indeed, some new programming languages have reduced this. But later we will see that you can do more interesting things in the constructor also.

It is a convention to start each property name with a capital letter, but to make parameter names start lower case. This also helps us to see which we are referring to when they have similar names.

A *class* can be thought of as a *template* that defines a type (Rectangle) and from which you can create multiple objects (also known as *instances)* each having its own copy of the properties containing its own individual values.

The *constructor* is the function that is used to create a new instance of that type (in this case to create a new Rectangle). Its parameters specify the values that you must provide to create an instance, and in the body of the constructor these parameters are used to set up the individual properties. (Many people comment that it looks a bit wasteful copying each parameter into a similarly-named property, but you'll soon get used to it. Some programming languages have side-stepped this apparent repetition.)

Notice how, unlike every other function you've written in C#, the constructor has no return type specified, not even `void`. This is the only place where this pattern occurs and helps identify the code

as a constructor. The other clue is that the name of the constructor must always be the same as the class that it constructs.

Use the same patterns to create a class called EquilateralTriangle, holding the properties needed to draw a triangle.

Q7 [Answer on EAD]	**Paste your code for the `EquilateralTriangle` class.**

Now we need to modify the code in Form1 to create a new instance of each class as needed, with the appropriate values for its properties, and then pass that object to the function for drawing it. The code below shows the code changes required for drawing a rectangle, you will need to figure out equivalent changes for triangle:

```
private void Form1_MouseClick(object sender, MouseEventArgs e)
{

    [some code not shown]

    else if (selectedItem == "Draw Rectangle")
    {
        var rec = new Rectangle(turtleX, turtleY, 100, 50);
        DrawRectangle(rec);
    }
}

private static void DrawRectangle(Rectangle rec)
{
    Turtle.ShowTurtle = false;
    Turtle.PenSize = 2;
    Turtle.Angle = 0;  //Always start from North
    Turtle.X = rec.XOrigin;
    Turtle.Y = rec.YOrigin;
    for (int i = 0; i < 2; i++)
    {
        Turtle.Forward(rec.Height);
        Turtle.Rotate(90);
        Turtle.Forward(rec.Width);
        Turtle.Rotate(90);
    }
}
```

Notice first, within the `Form1_MouseClick` function the use of `new` to create a new instance of Rectangle, passing in the required parameters. If you right click on the word `Rectangle` and select **Go To Definition** you will be taken to the *constructor* on the `Rectangle` class. And we assign this new instance to a variable `rec`, just as we would any existing type such as a string or an integer.

We have also changed the `DrawRectangle` function: instead of taking multiple parameters it now just takes a single parameter of type `Rectangle`. Inside the function we get (read) the property values we need using 'dot syntax' e.g. `rec.Height`.

You might well have already used dot syntax on standard C# types (e.g. `.Length` on a string). That's because C# (and all .NET languages) is a fully object-oriented language. You have actually been using *some* OOP patterns, unconsciously perhaps, from when you started writing C# - but now you are starting to use OOP consciously, creating your own object classes. ('Class' by the way is really a synonym for 'type' in most contexts).

Q8 [Answer on EAD] Paste in your equivalent code changes for drawing the EquilateralTriangle.

Running the code should produce the same results as before.

So, far then, we haven't really gained any benefit from introducing our own purpose-design object classes. But the benefits will become more apparent as we proceed.

Exercise 4. Adding behaviour to the object

So far we have used objects only for the purpose of holding multiple pieces data, all related but potentially of different types, in a single *instance* of a custom data type. Properly, though, object instances don't just have properties (also known as *data* and collectively representing the object's *state*) they have *methods* (also known as *functions* and collectively representing the object's *behaviour*). In fact, in real OOP the *behaviour* of objects is considered the more important of the two things.

Another way of putting this is that objects have two types of *responsibility*:

- What they *know* (or their 'know-whats'), represented by their properties.
- What they *know how to do* (or their 'know-how-tos'), represented by their methods.

The idea that objects have both types of responsibility is known as *encapsulation* - an object encapsulates all the responsibilities associated with the thing that it represents. Thus a Rectangle object should know everything that we need to known about a rectangle (for the purpose of the application we are developing, that is) and it should know how to do everything that we might want to do to a rectangle.

We'll start this process by transferring the responsibility for drawing individual objects:

Start by creating a new method (function) within the Rectangle class as follows. (The terms 'function' and 'method' are synonymous for many purposes – but they are more commonly referred to as methods in OOP). The body (implementation) of this method can be copied from the `DrawRectangle` function in Form1, but note the differences (listed below the code):

```
public class Rectangle
{
    //Properties
    private float XOrigin { get; set; }
    private float YOrigin { get; set; }
    private float Width { get; set; }
    private float Height { get; set; }

    //The 'Constructor'
    public Rectangle(float xOrigin, float yOrigin, float width, float height)
    { … }

    public void Draw()
    {
        Turtle.ShowTurtle = false;
        Turtle.PenSize = 2;
        Turtle.Angle = 0;
        Turtle.X = XOrigin;
        Turtle.Y = YOrigin;
        for (int i = 0; i < 2; i++)
        {
            Turtle.Forward(Height);
            Turtle.Rotate(90);
            Turtle.Forward(Width);
            Turtle.Rotate(90);
        }
    }
}
```

These are the important differences:

- Draw is `public` - which means that it can be accessed by code outside the class
- It does not need to take a `Rectangle` parameter, because it has *direct access* to the properties defined in the class.
- It does not have the keyword `static`. This is because we want `Draw` to operate on a specific instance of the class, not on the class as a whole.

Try *temporarily* adding the `static` keyword to `Draw`, you will find that the properties (which always belong to a specific instance) can no longer be accessed.

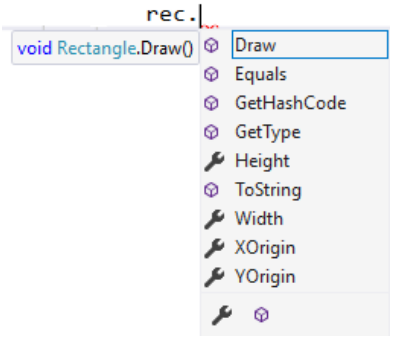
Q9 [Answer on EAD] **What compile error messages appear within the Draw method?**

Also because the properties of `Rectangle` are now accessed *only* by the Draw method on that class, we can make them fully private. This is an example of the principle of *Information Hiding* - but we will expand on that idea with a better example in a later exercise.

Now change the code in Form1 so that instead of calling the previous, local, `DrawRectangle` method (*which you should now delete*), it calls the Draw method*:*

```
else if (selectedItem == "Draw Rectangle")
{
    var rec = new Rectangle(turtleX, turtleY, 100, 50);
    rec.Draw();
}
```

Notice that we are invoking the new method using the *dot-syntax* again, only this time, the brackets after the name Draw indicate that it is a method (function) not a property. Also, in the auto-complete, you will see that properties have a small spanner icon next to them and methods don't:



(We'll find out where the other methods – such as Equals and ToString - come from later).

This 'dot syntax' combined with auto-complete is very helpful to the programmer when writing code. It means that many parts of your code will adopt a 'noun-verb' pattern: the instance of an object (held in a variable) is the noun, and the method that you invoke on that instance is the verb.

You might argue that we have gained only a small benefit from a large change - our program isn't yet doing anything new as a result. But we shall see that this idea of giving objects methods as well as properties - or *behaviour* as well as *state* – opens the way for more powerful techniques.

Make the equivalent changes for EquilateralTriangle

Q10 [Answer on EAD] Paste in the sections of code you changed, equivalent to those shown above for Rectangle.

Now, notice that the Rectangle class and the `EquilateralTriangle` class both define a method called `Draw()` - which has the same 'signature' in each case, meaning the same name, the same return type (`void` in this case) and the same parameters (none here). However, the two *implementations* are very different – apart from a few lines of code that are the same, a duplication that we will remove shortly.

Exercise 5. Introducing Polymorphism

Arguably the biggest benefit of moving behaviour into objects comes from *Polymorphism,* another of the key principles of OOP. Here is a definition for polymorphism:

> "Where two of more objects have a property or method with the same signature (name, return type, and, for a method, parameters), even though the implementations are different, then it is possible to access that property or method on the object, *without having two know the specific type of that object.*"

In the context of our drawing program, `Rectangle` and `EquilateralTriangle` both have a method called `Draw`, which has the same signature, though the implementation (how it is drawn) are quite different. They also have two properties (`XOrigin`, `YOrigin`) in common.

Therefore, if we had a collection (which could be a List or an Array) of shapes, made up of some Rectangles, some EquliateralTriangles, and potentially other forms of shape to be defined in future, then, with polymorphism, we could just call the Draw method on each shape without having to know that the first shape is a Rectangle, the second a EquilateralTriangle and so on. We could then add further methods such as Move, or Resize, and again call those on each member.

Think of how this works in PowerPoint, say. You can create a variety of shapes from the shapes menu. Then you can select multiple shapes, of different types, and then move them, grow them, colour them in one go, even though each form of shape has to do different things in response to those instructions.

First, how can we create a collection of disparate shapes? We can create an ordinary `ArrayList` and add any type of object into it, and then in a new method `DrawAll()` we could iterate over each shape in the list and call its `Draw()` method. Try making the following changes to the code in `Form1` (you will need to add `using System.Collections;` at the top to use an `ArrayList`):

```
private ArrayList shapes = new ArrayList();

private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    //Transform windows coordinates to Turtle coordinates
    float turtleX = e.X - Width / 2 + 8;
    float turtleY = Height / 2 - e.Y - 19;
    string selectedItem = (string) comboBox1.SelectedItem;
    if (selectedItem == "Draw Triangle") //We will add more options later
    {
        var tri = new EquilateralTriangle(turtleX, turtleY, 50);
        shapes.Add(tri);
        tri.DrawTriangle(); ;
    }
    else if (selectedItem == "Draw Rectangle")
    {
        var rec = new Rectangle(turtleX, turtleY, 100, 50);
        shapes.Add(rec);
        rec.Draw();
    }
}

public void DrawAll()
{
    Turtle.Dispose();  //First clear all Turtle tracks to start afresh
    foreach (var shape in shapes)
    {
        shape.Draw();
    }
}
```

Note the use of `Turtle.Dispose()` so we always create the drawing from a clean sheet.

But your code won't compile!

Q11 [Answer on EAD] What compile error message do you get?

The C# compiler is saying that there is no method named Draw on every object: after all you could be adding strings, Booleans, and integers into the list of shapes – try it!

(If we were using a dynamically typed language such as Python or JavaScript, we could get away with this. The system wouldn't check whether each object *actually* has a `Draw` method until you get to it at runtime - if it encounters an object that does not have a Draw method you get a runtime error. This can sometimes be more convenient, but the advantage of static typing is that you get more checking up front).

The solution is to define a new type called Shape, which does have a Draw method, and then specify that both Rectangle and EquilateralTriangle (and any new shape type we define) are in fact Shapes. So here's the new type:

```
namespace OOPDraw
{
    public interface Shape
    {
        void Draw();
    }
}
```

Notice that the new type is not a class - it is an *interface*. An interface defines the member signatures (*member* is a term that covers both properties and methods) but does not provide actual implementations. You can add this with the same **Add > New Item > Class** template - but over-writing the default code - or you can use the dedicated **Add > New Item > Interface** template.

So we need to declare that both Rectangle and EquilateralTriangle as well as having their own specific types *are also* of type Shape. This would require them both to provide an implementation of the Draw method defined on Shape - which, fortunately, they already have. We do this just by adding `: shape` ('implements Shape') after the class declaration in each file:

```
public class Rectangle : Shape
{

public class EquliateralTriangle : Shape
{
```

We can now change our list of shapes to be specifically a list containing instances of type `Shape`. (You'll need to change the `using` statement as shown). The `DrawAll()` method that you added above should now compile, because the code knows that anything that implements the type `Shape` has got a `Draw()` method. So now instead of calling the Draw method individually on the newly-created instance, we'll just call `DrawAll()` at the end of the `Form1_MouseClick` method instead:

```
using System.Collections.Generic;
using System.Windows.Forms;

namespace OOPDraw
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private List<Shape> shapes = new List<Shape>();

        private void Form1_MouseClick(object sender, MouseEventArgs e)
        {
            //Transform windows coordinates to Turtle coordinates
            float turtleX = e.X - Width / 2 + 8;
            float turtleY = Height / 2 - e.Y - 19;
            string selectedItem = (string) comboBox1.SelectedItem;
            if (selectedItem == "Draw Triangle") //We will add more options later
            {
                var tri = new EquilateralTriangle(turtleX, turtleY, 50);
                shapes.Add(tri);
                [code removed]
```

```
            }
            else if (selectedItem == "Draw Rectangle")
            {
                var rec = new Rectangle(turtleX, turtleY, 100, 50);
                shapes.Add(rec);
                [code removed]
            }
            DrawAll();
        }

        public void DrawAll()
        {
            foreach (var shape in shapes)
            {
                shape.Draw();
            }
        }
    }
}
```

Prove to yourself that your application still runs correctly.

Now let's make things more interesting. Define a new method `MoveTo` on Shape:

```
namespace OOPDraw
{
    public interface Shape
    {
        void Draw();

        void MoveTo(float x, float y);
    }
}
```

Building the code will now give us a compile errors.

Q12 [Answer on EAD] What compile error messages do you get?

The reason is that neither `Rectangle` nor `EquilateralTriangle` provides a `MoveTo` method. We can rectify this by adding an implementation into *both* those classes, which, unlike the `Draw` method, can be the same for each class. (Shortly we'll see how to avoid this duplication of code, but bear with it for now):

```
public void MoveTo(float x, float y)
{
    XOrigin = x;
    YOrigin = y;
}
```

We will first of all add the ability to move only the most recently-created object - later we'll extend this to allow us to move (or change the size of) any of the object's drawn:

```
private List<Shape> shapes = new List<Shape>();

private Shape mostRecent;

private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    //Transform windows coordinates to Turtle coordinates
    float turtleX = e.X - Width / 2 + 8;
    float turtleY = Height / 2 - e.Y - 19;
    string selectedItem = (string) comboBox1.SelectedItem;
    if (selectedItem == "Draw Triangle") //We will add more options later
    {
        var tri = new EquilateralTriangle(turtleX, turtleY, 50);
        shapes.Add(tri);
        mostRecent = tri;
    }
    else if (selectedItem == "Draw Rectangle")
    {
        var rec = new Rectangle(turtleX, turtleY, 100, 50);
        shapes.Add(rec);
        mostRecent = rec;
    }
    DrawAll();
}
```

Now go back to the Designer view of Form1 and edit the Items on the ComboBox to add **Move Shape**. Then handle that option in the `Form1_MouseClick` method as follows:

```
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    //Transform windows coordinates to Turtle coordinates
    float turtleX = e.X - Width / 2 + 8;
    float turtleY = Height / 2 - e.Y - 19;
    string selectedItem = (string) comboBox1.SelectedItem;
    if (selectedItem == "Draw Triangle") //We will add more options later
    {
        var tri = new EquilateralTriangle(turtleX, turtleY, 50);
        shapes.Add(tri);
        mostRecent = tri;
    }
    else if (selectedItem == "Draw Rectangle")
    {
        var rec = new Rectangle(turtleX, turtleY, 100, 50);
        shapes.Add(rec);
        mostRecent = rec;
    }
    else if (selectedItem == "Move Shape")
    {
        mostRecent.MoveTo(turtleX, turtleY);
    }
    DrawAll();
}
```

Verify that after having drawn a shape you can move it around by selecting the Move Shape option and clicking. Previously drawn shapes remain in position.

Exercise 6. Introducing Inheritance

One of the most important principles in programming is *The DRY Principle* : Don't Repeat Yourself. Repeated code is not just wasteful in terms of space, it adds to the risk that two pieces of code that where intended to be identical, instead diverge over time - because one gets updated and the other overlooked. In procedural programming we could fix this by making two functions delegate the execution of the logic to a third function called by both.

In OOP, where the identical code exists on classes that have something in common (as Rectangle and EquilateralTriangle do) then we can use another, elegant, way to eliminate the duplication: *inheritance.* To do this we are going to change our definition of Shape from an *Interface* to an *Abstract class.* In many respects these concepts are similar, though the syntax is slightly different. However, as we shall see, abstract classes have the option to provide implementation code, as well as defining the member signature, where the implementation would be the same. Make these changes to `Shape`:

```
namespace OOPDraw
{
    public abstract class Shape
    {
        public abstract void Draw();

        public abstract void MoveTo(float x, float y);

    }
}
```

And add the word `override` to the implementation of each of these methods in *both* `Rectangle`, and `EquilateralTriangle`:

```
public override void Draw()
{
  . . .
}

public override void MoveTo(float x, float y)
{
  . . .
}
```

The code should run exactly as before: Shape defines the methods abstractly, the two other classes define them concretely (with an implementation).

Now we'll start moving common implementation code 'up' from `Rectangle` and `EquilateralTriangle` into `Shape`. Start with the two properties, `XOrigin` and `YOrigin`, then replace the abstract definition of `MoveTo` on the class `Shape` with a copy of the implementation from either of the other two shape classes, but removing the word `override`, so that `Shape` ends up like this:

```
namespace OOPDraw
{
    public abstract class Shape
    {
        //Properties
        protected float XOrigin { get; set; }
        protected float YOrigin { get; set; }

        //Abstract methods
        public abstract void Draw();

        //Concrete methods
        public void MoveTo(float x, float y)
        {
            XOrigin = x;
            YOrigin = y;
        }
    }
}
```

Notice that in moving the properties we have changed them from being `private` to `protected` - which can be thought of as being between `public` and `private`. It means that the properties can be accessed within the class, *and by its sub-classes*, but not from anywhere else.

The two properties and one method moved 'up' into `Shape` should both be deleted from `Rectangle` and `EquliateralTriangle`.

Confirm that your program still runs correctly.

Q13 [Answer on EAD] Paste in a screenshot showing that you can still draw rectangles and triangles.

Because all three are now classes, we say that Shape is the *super-class* of `Rectangle` and `EquliateralTriangle`, and that they, in turn, are *sub-classes* of Shape. We also say that they *inherit* member *definitions* and, in some cases, member *implementation* from their super-class.

We *could* have multiple levels of inheritance. For example, we could create `RegularPolygon` inheriting from `Shape`, and then make `EquilateralTriangle`, `Square`, and `Pentagon`, say, inherit from `RegularPolygon` - but not `Rectangle`, which would continue to be a *direct sub-class* of `Shape`.

So we have removed some duplication from our code. But if you look carefully there are still *some* common lines of code between `Rectangle` and `EquliateralTriangle`: in their implementations of the Draw() method, and in their constructors. Good programming means becoming *obsessive* about the DRY principle, so we are going to use another technique - *delegation* - to fix it.

First, we will add a constructor to the Shape class. An abstract class is never instantiated directly, only through its sub-classes, but the value in defining a constructor is that we can provide some common logic to be re-used by the constructors in the sub-classes.

(By the way, it is possible to have a *concrete super-class* – one that can be instantiated by itself. But some OOP practitioners dislike this pattern, and it is better at this stage to think of super-classes as abstract and sub-classes as concrete).

Here's the new code for `Shape`:

```
public abstract class Shape
{
    protected float XOrigin { get; set; }
    protected float YOrigin { get; set; }

    //The 'Constructor'
    public Shape(float xOrigin, float yOrigin)
    {
        XOrigin = xOrigin;
        YOrigin = yOrigin;
    }
```

Then remove those two lines of code from the constructors of the two sub-classes and instead specify that those constructors 'extend the base constructor' (the constructor of their super-class) as follows (note that lines are broken only to fit this document - the new code can go on the end of the line above it):

```
public Rectangle(float xOrigin, float yOrigin, float width, float height)
                                                        : base(xOrigin, yOrigin)
{
    [code removed]
    Width = width;
    Height = height;
}

public EquilateralTriangle(float xOrigin, float yOrigin, float sideLength)
                                                        : base(xOrigin, yOrigin)
{
    [code removed]
    SideLength = sideLength;
}
```

For the duplication in the `Draw` method we will extract the common code into a new method on the super-class, `Shape` :

```
protected void ResetTurtle()
{
    Turtle.ShowTurtle = false;
    Turtle.PenSize = 2;
    Turtle.Angle = 0;  //Always start from North
    Turtle.X = XOrigin;
    Turtle.Y = YOrigin;
}
```

And replace the equivalent code in the two sub-classes with a call to this method e.g. for `Rectangle`:

```
public override void Draw()
{
    ResetTurtle();
    for (int i = 0; i < 2; i++)
    {
        Turtle.Forward(Height);
        Turtle.Rotate(90);
        Turtle.Forward(Width);
        Turtle.Rotate(90);
    }
}
```

Make sure you do the same for `EquilateralTriangle`.

Q14 [Answer on EAD] Paste your new code for the Draw method on EquilateralTriangle

Notice that the new `ResetTurtle` method on `Shape` is neither `private` not `public`, but `protected` - the meaning of which is somewhere between the two. It means that the method can be accessed within the `Shape` class, and also within its sub-classes, but not external code (to which it appears private). As a general rule, members should not be exposed more than they strictly need to be. This principle of keeping the workings of objects relatively hidden will be expanded in the next exercise.

First, a couple of reflections on inheritance.

Since an abstract class allows us to do everything we could do with an interface, and also provide the option of inherited implementations, why would we not always use an abstract class instead of an interface? What unique benefit does the concept of an interface offer? The biggest advantage of interfaces is that in C# (and VB) a class may only inherit directly from one super-class, but it may *also* implement one or more interfaces. Some languages, such as Python, permit a class to inherit from more than one superclass. So, in our application, we might define a shape called `Lozenge`, which inherits directly from the concrete class `Rectangle`, and also inherit directly from a separate abstract class `ShapeWithRoundedCorners`, say. Although this appears to offer a lot of flexibility, it can lead to some quite complex problems, and many OOP practitioners dislike this idea of multiple inheritance. In C# or VB we could however make all our shapes inherit from `Shape`, but also define a series of interfaces - `IColourable`, `IFillable`, `IRotatable`, say - and individual sub-classes of Shape could implement whichever of those interfaces were applicable. Each interface would define the method(s) and/or properties that would be required. An interface can therefore be thought of as a *role* than an object can fulfil - such that that object could be processed alongside different objects that all fulfil the same role.

(Incidentally, the three example interfaces listed above all have names with the prefix '`I`' - this is a widespread convention in OOP, but it is not a requirement, as we saw earlier when `Shape` started as an interface before being changed to an abstract class.)

It is also worth saying that inheritance is frequently over-used. Every time you see code that is duplicated in two or more classes it can be tempting to make them inherit from a common super-class - when in fact you could just delegate the implementation to something else (we'll see an example of this shortly). The most practical rule is: only define a super-class where the name of that super-class has a clear and obvious meaning: Shape as a super-class of Rectangle and Triangle passes this test.

A good example of the dilemma is the known as the 'Circle-Ellipse Problem' in OOP. Circles and Ellipses are defined by the same algebraic formula: $ax^2 + by^2 = k$, but where a and b are set to 1 for a

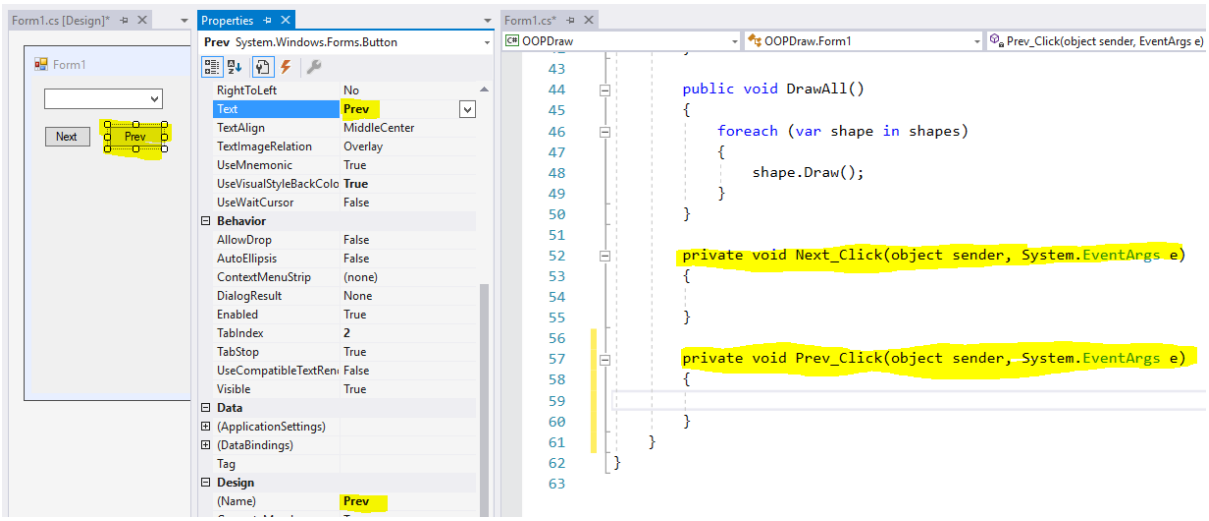
circle (and k would then be the square of the radius). But if you did this in our drawing program, would you make `Circle` inherit from `Ellipse`? Or *vice versa*? It turns out that there are problems with both of those approaches - such as what happens if the user attempts to stretch the shape in one axis only - and many academic papers have been written about it. Another option would be to make `Circle` and `Ellipse` both inherit from an abstract class called, say, `RoundishShape` - but does this pass the common sense test? Best advice: if you can't see a clear and problem-free pattern for inheritance, don't use it. Leave them both inheriting only from `Shape` (which is not controversial) and then make `Ellipse`, but not `Circle`, implement the `IStretchable` interface.

We will next use this new capability of inheritance to add some more common features, to achieve our next goal of being able to move any of the shapes already drawn, and to learn another OOP principle: information hiding.

Exercise 7. Introducing Information Hiding

The previous exercise allowed us to draw and then move a shape, but we would like to be able to move any shape already drawn. To do this we need a mechanism to select any single shape. Ideally, we'd like just to click on it with the mouse - as you would in PowerPoint, say - but this would be quite a difficult programming challenge at this stage. (At the end of this paper you will find several suggestions as to how our basic drawing program could be expanded or improved if you would like to take it further). So instead we'll add two buttons, **Next** and **Prev**(ious) to allow the user to cycle through the shapes.

In each case add the button via the form designer. Right-click on each button to view its properties, and change their **Text** *and* **Name** properties. *After doing that,* double click on each button, to create the code-behind method, so you end up with this:



Then we'll add code into Form1 to hold, and modify, an index to the 'active' (or current) shape, and also provide a method to access the ActiveShape from within the list of shapes.

```
private int activeShapeNumber = 0;

private Shape ActiveShape()
{
    return shapes[activeShapeNumber]; //List elements can be accessed like an array
}

private void Next_Click(object sender, System.EventArgs e)
{
    activeShapeNumber = activeShapeNumber + 1;
    if (activeShapeNumber >= shapes.Count) activeShapeNumber = 0;
}

private void Prev_Click(object sender, System.EventArgs e)
{
    activeShapeNumber = activeShapeNumber - 1;
    if (activeShapeNumber < 0) activeShapeNumber = shapes.Count - 1;
}
```

The second line in the body of each of the `_Click` methods is so that we can cycle through all the shapes - going from the last back to the first shape or *vice versa*.

We can now modify the Form1_MouseClick method to use this new capability:

```
private Shape mostRecent; //Delete this line

private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    //Transform windows coordinates to Turtle coordinates
    float turtleX = e.X - Width / 2 + 8;
    float turtleY = Height / 2 - e.Y - 19;
    string selectedItem = (string) comboBox1.SelectedItem;
    if (selectedItem == "Draw Triangle") //We will add more options later
    {
        var tri = new EquilateralTriangle(turtleX, turtleY, 50);
        shapes.Add(tri);
        mostRecent = tri; //Delete this line
        activeShapeNumber = shapes.Count - 1; //i.e. the shape just added
    }
    else if (selectedItem == "Draw Rectangle")
    {
        var rec = new Rectangle(turtleX, turtleY, 100, 50);
        shapes.Add(rec);
        mostRecent = rec; //Delete this line
        activeShapeNumber = shapes.Count - 1; //i.e. the shape just added
    }
    else if (selectedItem == "Move Shape")
    {
        ActiveShape().MoveTo(turtleX, turtleY);
    }
    DrawAll();
}
```

Run the program, create some shapes, and then select Move. You can then use the Next and Prev to select any shape and then click on screen to move it. But you will find that this isn't very user-friendly.

Q15 [Answer on EAD] Why is the program not user-friendly?

Real drawing programs provide visual feedback on which shape is currently active or 'selected'. This might be signified by a change in colour, or by the temporary addition of 'handles' to the shape. The simplest mechanism we would implement is to draw the selected shape in bold - by temporarily increasing the line width. So we could give every shape a `LineWidth` property (by adding it to the `Shape` class), make use of this in the `Draw` method to control the Turtle's `PenWidth`, and then have the code in `Form1` increase the `LineWidth` on the active object. So modify `Shape` to:

```
public abstract class Shape
{
    protected float XOrigin { get; set; }
    protected float YOrigin { get; set; }
    public float LineWidth { get; set; }

    //The 'Constructor'
    public Shape(float xOrigin, float yOrigin)
    {
        XOrigin = xOrigin;
        YOrigin = yOrigin;
    }

    //Abstract methods
    public abstract void Draw();

    //Concrete methods
    public void MoveTo(float x, float y)
    {
        XOrigin = x;
        YOrigin = y;
    }

    protected void ResetTurtle()
    {
        Turtle.ShowTurtle = false;
        Turtle.PenSize = LineWidth;
        Turtle.Angle = 0;  //Always start from North
        Turtle.X = XOrigin;
        Turtle.Y = YOrigin;
    }
}
```

And on Form1.cs:

```
private void Next_Click(object sender, System.EventArgs e)
{
    ActiveShape().LineWidth = 2;
    activeShapeNumber = activeShapeNumber + 1;
    if (activeShapeNumber >= shapes.Count) activeShapeNumber = 0;
    ActiveShape().LineWidth = 4;
}

private void Prev_Click(object sender, System.EventArgs e)
{
    ActiveShape().LineWidth = 2;
    activeShapeNumber = activeShapeNumber - 1;
    if (activeShapeNumber < 0) activeShapeNumber = shapes.Count - 1;
    ActiveShape().LineWidth = 4;
}
```

The problem with this pattern is that it ties the code in `Form1` to a particular technique for rendering a selected object – modifying the line width. If, later, we decide to change the visual rendering of a selected object, perhaps to use a different colour, or add the handles, then we not only have to change the `Draw` methods on the shapes but also the code in `Form1`. Good OOP design aims to minimise the impact of changes in one part of the object model on code elsewhere. We do this

through the technique of *information hiding*. Instead of exposing the `LineWidth` property outside the shape, we make it `private`, and, instead, offer `public` methods that convey the behaviour that is really required by the external code: `Select` and `Unselect`:

```
public abstract class Shape
{
    protected float XOrigin { get; set; }
    protected float YOrigin { get; set; }
    private float LineWidth { get; set; }

    //The 'Constructor'
    public Shape(float xOrigin, float yOrigin)
    {
        XOrigin = xOrigin;
        YOrigin = yOrigin;
    }

    //Abstract methods
    public abstract void Draw();

    //Concrete methods
    public void MoveTo(float x, float y)
    {
        XOrigin = x;
        YOrigin = y;
    }

    public void Select()
    {
        LineWidth = 4;
    }

    public void Unselect()
    {
        LineWidth = 2;
    }

    protected void ResetTurtle()
    {
        Turtle.ShowTurtle = false;
        Turtle.PenSize = LineWidth;
        Turtle.Angle = 0;  //Always start from North
        Turtle.X = XOrigin;
        Turtle.Y = YOrigin;
    }
}
```

And the code in `Form1` can be changed to:

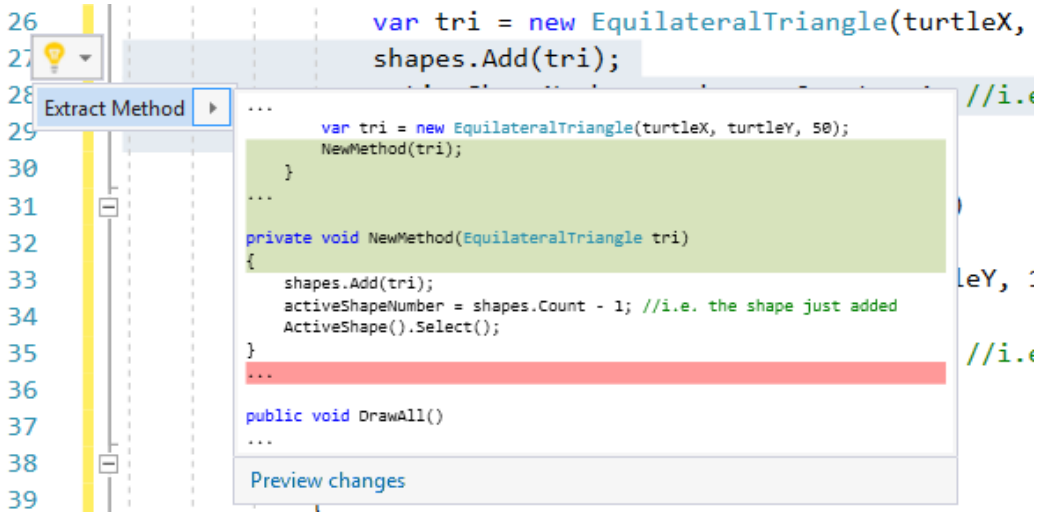
```
private int activeShapeNumber = 0;

private Shape ActiveShape()
{
    return shapes[activeShapeNumber]; //List elements can be accessed like an array
}

private void Next_Click(object sender, System.EventArgs e)
{
    ActiveShape().Unselect();
    activeShapeNumber = activeShapeNumber + 1;
    if (activeShapeNumber >= shapes.Count) activeShapeNumber = 0;
    ActiveShape().Select();
    DrawAll();
}

private void Prev_Click(object sender, System.EventArgs e)
{
    ActiveShape().Unselect();
    activeShapeNumber = activeShapeNumber - 1;
    if (activeShapeNumber < 0) activeShapeNumber = shapes.Count - 1;
    ActiveShape().Select();
    DrawAll();
}
```

Note that we need to call `DrawAll` after any selection has been changed, to refresh the display.

One small inconvenience remains. It would be more user-friendly if the most recently-added shape was automatically selected. We can easily do this thus:

```
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    //Transform windows coordinates to Turtle coordinates
    float turtleX = e.X - Width / 2 + 8;
    float turtleY = Height / 2 - e.Y - 19;
    string selectedItem = (string)comboBox1.SelectedItem;
    if (selectedItem == "Draw Triangle") //We will add more options later
    {
        var tri = new EquilateralTriangle(turtleX, turtleY, 50);
        shapes.Add(tri);
        activeShapeNumber = shapes.Count - 1; //i.e. the shape just added
        ActiveShape().Select();
    }
    else if (selectedItem == "Draw Rectangle")
    {
        var rec = new Rectangle(turtleX, turtleY, 100, 50);
        shapes.Add(rec);
        activeShapeNumber = shapes.Count - 1; //i.e. the shape just added
        ActiveShape().Select();
    }
```

But, as you have hopefully noticed, we now have *three* lines essentially the same in two blocks of code: which clearly violates the DRY principle. We should *refactor* this code and extract a common method. In fact, Visual Studio will do some of the work for us. Select the first occurrence of the three lines, right click and select **Quick Actions**, then **Extract Method**:



Then immediately rename the new method to `AddShape`. Go to the newly created method and change the type of the parameter to Shape, so it looks like this:

```
 private void AddShape(Shape shape)
{
    shapes.Add(shape);
    activeShapeNumber = shapes.Count - 1; //i.e. the shape just added
    ActiveShape().Select();
}
```

And then make sure the three repeated lines are replaced by a call to this method. In fact you can even coalesce the two remaining lines into one, thus:

```
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    //Transform windows coordinates to Turtle coordinates
    float turtleX = e.X - Width / 2 + 8;
    float turtleY = Height / 2 - e.Y - 19;
    string selectedItem = (string)comboBox1.SelectedItem;
    if (selectedItem == "Draw Triangle") //We will add more options later
    {
        AddShape(new EquilateralTriangle(turtleX, turtleY, 50));
    }
    else if (selectedItem == "Draw Rectangle")
    {
        AddShape(new Rectangle(turtleX, turtleY, 100, 50));
    }
    else if (selectedItem == "Move Shape")
    {
        ActiveShape().MoveTo(turtleX, turtleY);
    }
    DrawAll();
}
```

Run the program and try this out. There is a bug.

Q16 [Answer on EAD] **What is the bug?**

To fix this we need to unselect the current shape immediately before adding the new one. Fortunately, but only because of our recent refactoring, we now need make this change in one place only:

```
private void AddShape(Shape shape)
{
    ActiveShape().Unselect();
    shapes.Add(shape);
    activeShapeNumber = shapes.Count - 1; //i.e. the shape just added
    ActiveShape().Select();
}
```

Run the program. We have fixed that bug, but created a different bug in the process (not uncommon in software development!)

Q17 [Answer on EAD] What error arises, and why?

We need to put in a test to see if this is the very first shape being added, in which case there will be no ActiveShape to unselect. So:

```
private void AddShape(Shape shape)
{
    if (shapes.Count > 0) //i.e. this isn't the first shape
    {
        ActiveShape().Unselect();
    }
    shapes.Add(shape);
    activeShapeNumber = shapes.Count - 1; //i.e. the shape just added
    ActiveShape().Select();
}
```

Run the program and verify that now only the most recently-added shape is rendered in bold, and that you can cycle through the drawn shapes using Next and prevous.

Q18 [Answer on EAD] Paste in a screenshot showing one of several shapes selected.

Exercise 8. Adding a Resize action

We have the ability to move shapes we have previously created, now we want to add another action to resize them. (At the end of this workbook you will find suggestions for further actions you could add, such as to rotate a shape, or recolour it).

We'll start by defining an abstract method on `Shape`:

```
public abstract void Resize(float x, float y);
```

And a specific implementation on the two concrete sub-classes, starting with `Rectangle`, where we use the parameters to change the Width and Height of the rectangle.

```
public override void Resize(float x, float y)
{
    Width = x;
    Height = y;
}
```

An `EquilateralTriangle` can't be resized independently in two dimensions so we will just ignore the y parameter and use x to determine the new side length:

```
public override void Resize(float x, float y)
{
    //Ignore Y
    SideLength = x;
}
```

We can also add a **Resize Shape** menu option onto the form. Follow the same steps that you did for adding the **Move Shape** option previously. Then implement this in the `Form1_MouseClick` method, below the other options:

```
else if (selectedItem == "Resize Shape")
{
    ActiveShape().Resize(turtleX, turtleY);
}
```

Make these changes, run the program and try the new menu option on a shape. You should find that it resizes - but does it re-size as you expect.?

Q19 [Answer on EAD] What happens if you resize a triangle, and click somewhere low on the screen?

The problem is because Turtle coordinates are relative to the centre point of the screen and may therefore be positive or negative. It would be more natural if the resize was based on the position where you click *relative to the origin of the shape being resized.* The following code would fix that:

```
else if (selectedItem == "Resize Shape")
{
    var s = ActiveShape();
    s.Resize(Math.Abs(turtleX - s.XOrigin), Math.Abs(turtleY - s.YOrigin));
}
```

But this code would not compile, because the `XOrigin` and `YOrigin` properties on `Shape` are not public (we deliberately made them `protected`). We *could* make them public. The code would then compile and work, but we would be left with what experienced OO programmers call an 'anti-pattern' (sometimes also called a 'code smell'!): we are reading properties from an object solely in

order to use them in a call to a method on that *same* object. This logic should really be inside the Shape class. But how?

The fix is to put it in a new method with a slightly different name -`ResizeAbsolute`, say - and make this method delegate to the abstract `Resize` method, that is implemented on the sub-classes. So to `Shape` we add:

```
public void ResizeAbsolute(float turtleX, float turtleY)
{
    Resize(Math.Abs(turtleX - XOrigin), Math.Abs(turtleY - YOrigin));
}

public abstract void Resize(float x, float y);
```

and we call the new method from `Form1_MouseClick:`

```
else if (selectedItem == "Resize Shape")
{
    ActiveShape().ResizeAbsolute(turtleX, turtleY);
}
```

Make these changes. Then use the program to draw a simple representation of a house: a rectangle for the walls and a triangular roof immediately above it, of the same width.

Q20 [Answer on EAD] **Paste in a screenshot of your running program showing the house.**

Exercise 9. Adding a House symbol, using association and delegation

In the previous exercise you drew a house by drawing a separate rectangle and triangle. Now supposing that in your job you needed to add the same simple symbol for a house frequently, on different drawings, perhaps of different sizes: hardly a requirement for an architect, agreed, but possibly for a map-maker. It would be useful if we could draw the house symbol just like a more primitive shape.

One solution would be to create a new House class, inheriting from Shape and just define the Turtle movements for the complete symbol. Not a bad approach for such a simple symbol, maybe, but not really in keeping with our policy of minimising repetition.

So could House inherit from `Rectangle`, or `EquilateralTriangle` and then just add the further lines? If so, which one? Or, if your programming language supported *multiple inheritance* (Python, say) could House inherit from *both* `Rectangle` and `EquilateralTriangle`? Actually, none of these is a good idea, though it is a common mistake by people learning OOP.

The key thing to understand is that inheritance relationship is characterised by the relationship '*is a specific type of*' - sometimes abbreviated to just *'is a'*. Thus, 'a Rectangle *is a specific type of* Shape', or 'a Pentagon *is a specific type of* RegularPolygon'. It makes no sense to say that our 'House (symbol) *is a specific type of* Rectangle', nor of a triangle. It would be correct to say that the 'House (symbol) *has a* Rectangle and *has a* Triangle (for its roof)'. The '*has a'* relationship does not translate to inheritance, but to *association*.

Add a new class House to your project, with this code:

```
namespace OOPDraw
{
    public class House : Shape
    {
        private float Width { get; set; }
        private float WallHeight { get; set; }
        private Rectangle Walls { get; set; }
        private EquilateralTriangle Roof { get; set; }

        public House(float originX, float originY, float width, float wallHeight) :
base(originX, originY)
        {
            Width = width;
            WallHeight = wallHeight;
            Walls = new Rectangle(originX, originY, width, wallHeight);
            Roof = new EquilateralTriangle(originX, originY + wallHeight, width);
        }

    }
}
```

Notice that `House` *is a* `Shape`, but that, as well as its own basic properties , it *has a* property of type `Rectangle` named `Walls`, and a property of type `EquilateralTriangle` named `Roof`. We say that House has an *association to* those other objects. Notice also that constructor for `House`, creates a new `Rectangle` and `EquilateralTriangle` of the right size and position, and puts them into the `Walls` and `Roof` properties.

The code will not compile yet because we haven't yet implemented the abstract methods required by `Shape`. If you right-click on the `House` class name (where the green highlight is shown above) and invoke the **Quick Actions > Implement Abstract Class**, you will see the following 'skeleton code' added:

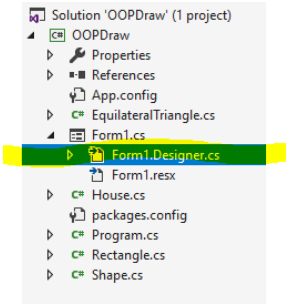
```
public override void Draw()
{
    throw new System.NotImplementedException();
}

public override void Resize(float x, float y)
{
    throw new System.NotImplementedException();
}
```

The code will now compile and we can replace the generated line `throw new System.NotImplementedException();` with a proper implementation when we are ready - or when we need to. Throwing the `NotImplementedException` by default is a very good practice. It means that if we ran the program and this method was called, execution would halt and we would know the reason. If we had just left the method empty we might not remember why the code was not doing what it should.

Implementing the `Draw` method for `House` is easy - we just *delegate* the problem to the associated objects:

```
public override void Draw()
{
    Walls.Draw();
    Roof.Draw();
}
```

Add a **Draw House** option into the ComboBox on the form. You *could* do this via the interactive Designer as before, but let's experiment with a different way of doing it this time: by directly editing the `Form1.Designer.cs` file:



```
29          private void InitializeComponent()
30          {
31              this.comboBox1 = new System.Windows.Forms.ComboBox();
32              this.Next = new System.Windows.Forms.Button();
33              this.Prev = new System.Windows.Forms.Button();
34              this.SuspendLayout();
35              //
36              // comboBox1
37              //
38              this.comboBox1.Items.AddRange(new object[] {
39              "Draw Triangle",
40              "Draw Rectangle",
41              "Draw House",
42              "Move Shape",
43              "Resize Shape"});
```

Also modify the `Form1_MouseEvent` method in `Form1.cs` (not the same file as `Form1.Designer.cs`) to handle this new option, in the same way that it handles **Draw Rectangle**, say.

Run the program generate multiple houses in different locations.

Q21 [Answer on EAD] Paste in a screenshot showing multiple houses.

But notice that each new House is not being rendered in bold.

Q22 [Answer on EAD] Can you figure out why not?

Notice also that Move does not appear to do anything - even though this is an inherited implementation.

Q23 [Answer on EAD] What happens if you draw a house, then select Resize and click somewhere? Is this expected?

To solve the first problem we need to make the `Select` and `Unselect` methods also delegate to the associated objects - but just for a `House`. The technique for this is to *override* the inherited implementation, with an implementation specific to that sub-class. Previously we have seen that an `abstract` superclass can define an `abstract` method, for which its sub-classes *must* provide their own implementation; or a concrete method where the super-class provides the implementation. We are now going to use a third option: where the super-class provides *an* implementation but permits the sub-classes to override this if they need to provide their own, more specialised, implementation. The keyword for this in C# is `virtual` (the equivalent in VB is more intuitive: `overridable`). So modify just the `Select` and `Unselect` methods in `Shape` to:

```
public virtual void Select()
{
    LineWidth = 4;
}

public virtual void Unselect()
{
    LineWidth = 2;
}
```

And then add new, overriding, implementations into just the `House` class:

```
public override void Select()
{
    Walls.Select();
    Roof.Select();
}

public override void Unselect()
{
    Walls.Unselect();
    Roof.Unselect();
}
```

Q24 [Answer on EAD] **What compile error message do you get if you *temporarily* remove the keyword `virtual` from either of the inherited methods on Shape?**

Run the program and prove that you can now select a house.

Q25 [Answer on EAD] **Paste in a screenshot.**

But **Move Shape** still won't work.

We need to do something similar for the `Move` method. First make it `virtual` in the super-class and then add this specific implementation into `House`.

```
public override void MoveTo(float x, float y)
{
    base.MoveTo(x, y);
    Walls.MoveTo(x, y);
    Roof.MoveTo(x, y + WallHeight);
}
```

The first line of this implementation calls the `base` version of the same method - meaning the default implementation provided on the superclass. This just sets the **`XOrigin`** and **`YOrigin`** for the `House`. If we didn't do this, then `Move` would possibly still work - but the `House` would now have coordinates that no longer related to the walls and roof - storing up trouble for the future! Notice also that the `Roof` is moved to a the new coordinates plus the **`WallHeight`** - just as it was when the `House` was created.

Implementing the `Resize` method in `House` involves similar ideas, but is surprisingly tricky:

```
public override void Resize(float x, float y)
{
    Width = x;
    var yDiff = y - WallHeight;
    WallHeight = y;
    Walls.Resize(x, y);
    Roof.Resize(x, 0);
    Roof.MoveBy(0, yDiff);
}
```

You'll also need to add the following new method into `Shape`, to provide relative movement:

```
public virtual void MoveBy(float x, float y)
{
    XOrigin += x;
    YOrigin += y;
}
```

Run the program and prove that you can now resize a whole house.

Q26 [Answer on EAD] **Paste in a screenshot showing several houses of different sizes.**

If you have time and interest, you might like to remove these implementations and figure them out from scratch - but it might take you quite a bit of debugging.

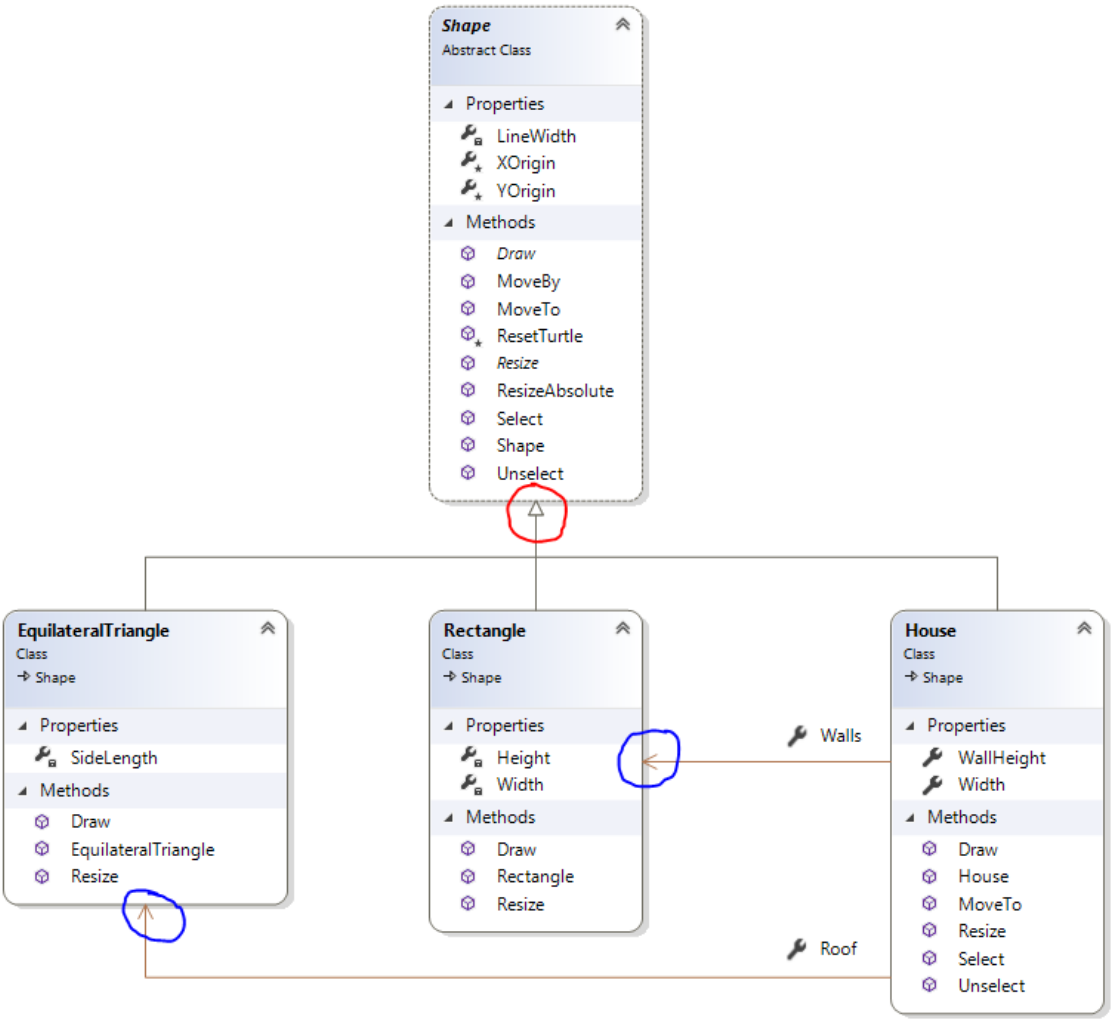
So you would be forgiven for thinking that it might have been simpler just to create `House` from scratch, specifying each line as a Turtle instruction directly. That might well be right for this simple example of a house symbol - but the general idea of delegating responsibilities to associated objects lies at the core of OOP, and you will certainly need the technique if you build more complex programs. It would become more obvious if you made the house a little more realistic, by adding a door and several windows, say - and this is one of the suggested optional exercises at the end of the worksheet.

Or think about the following scenario. On PowerPoint, like most real drawing programs, it is possible to select a number of drawn shapes and turn them into a single Group. Group is still a Shape, and offers all the same methods that other shapes do, but it delegates execution to the shapes that made up the group - which still exist as associated objects. We know this because it is always possible to 'un-Group' them later, back into their separate objects. You could even have a go at implementing the Group idea on this program. You will need all the techniques you have used in creating House, and more. The Group object will need to hold a List of its member shapes, for example.

One last point about association. The A-Level specification draws a distinction between two forms of association: *aggregation* and *composition*. The relationship between `House`, and the objects representing the walls and the roof is *composition*. The house *is composed of* the walls and roof (and possibly other shapes or properties). You could also say the House *owns* the walls and roof. If you were to delete the house (that could be another function to add to the program) then the `Rectangle` and `EquilateralTriangle` instances representing its walls and roof would be deleted with it.

Aggregation is where objects are associated, but the relationship is not one of ownership. There isn't an obvious example of aggregation for this Drawing program. But in a school management system, for example, we might have Student and Teacher objects, which would be associated via the relationship of 'tutor', say. But if either the Student or the Teacher were to leave the school, that would not, we hope, imply that the other would automatically be removed!

The diagram below shows all of the classes that we have developed in these exercises so far, together with the names of the properties and methods that each defines. It also shows the inheritance relationships and, separately, the associations. Notice the difference in the styles of the arrows: the arrow head indicating inheritance is circled in red pen, and the arrow indicating association is circled in blue pen. The *association* relationships also have a property indicating the nature or 'role' of the association, corresponding to the name of the property in code.


Shape
Abstract Class
Properties
LineWidth
XOrigin
YOrigin
Methods
Draw
MoveBy
MoveTo
ResetTurtle
Resize
ResizeAbsolute
Select
Shape
Unselect
EquilateralTriangle
Class
Shape
Properties
SideLength
Methods
Draw
EquilateralTriangle
Resize
Rectangle
Class
Shape
Properties
Height
Width
Methods
Draw
Rectangle
Resize
Walls
Roof
House
Class
Shape
Properties
WallHeight
Width
Methods
Draw
House
MoveTo
Resize
Select
Unselect

Exercise 10. Ideas for extending or improving OOPDraw

Hopefully this set of exercises has whetted your appetite for object-oriented programming. Though OOPDraw is simplistic at present, its basic design could be expanded to a fully-functional drawing application, or even a specialised Computer Aided Design package. If you fancy some real challenges, why not explore the following, and then add some more ideas of your own?

- Add further types of shape, such as `Circle`, including non-solid shapes such as `Line` and `Arrow`.
- Make the `House` class more complex, adding a door and windows for example, ensuring that these are moved and resized correctly when those methods are called on `House`.
- Add the ability to specify and change the colour and width of the lines
- Give `Shape` an `Orientation` property and add a method to `Rotate` the shape.
- Add the ability to multi-select shapes, and then to apply the same action to each shape selected in one go.
- Allow the selection to be turned into a `Group`, a single object that takes the place of the individual members and delegates responsibility for actions to them, mediated or modified as necessary
- Change the implementation of `Select` and `Unselect` to change the colour of the shape, or perhaps add 'handles'.
- Change the user 'gesture' for moving or resizing from 'point-and-click' to 'drag-and-drop', so that a shape moves or resizes continuously.
- Replace Turtle graphics with line-based drawing functions using the in-built .NET graphics library
- Allow already-drawn shapes to be selected by clicking on them instead of needing to cycle through the shapes with the **Next** and **Prev** buttons.
- Add simple animation to record movement of shapes and play back those same movements.