Polymorphism

In this paper, we look at the 4 underlying principles of object oriented programming:

- **Abstraction** – identifying the behaviours (stored in the methods) and data (stored in the fields) that define a class.

- **Encapsulation** – creating objects (that belong to an existing class) by encapsulating/bundling together the specific individual data and the methods that can act on that data.

- **Inheritance** – reorganising our classes into a hierarchy where the common fields and methods are put into a parent class, and leaving the unique fields and methods in the individual child classes.

- **Polymorphism** – if there is inheritance, a method in the parent class can be overridden by methods (with the same signature) in each and every child class.

Polymorphism

Using polymorphism we can declare subclasses which descend from the same base class, have the same fields, properties and methods, but which implement some of those methods differently. Imagine, for example, that you wish to construct a class structure for **polygons** that includes **circles**, **squares** and **triangles**. All of these figures would have some common fields, like location and size, and some common methods which they implement in the same way, like the properties, the constructor or a Move() method (implemented by modifying their x and y coordinates). They might have some methods however, that they implement differently. For example, these polygons might need to compute their areas. For squares, this value is (base * height); for circles, it is ($\pi * r^2$); for triangles, it is (0.5 * base * height). Each subclass needs a CalculateArea() method, but the code contained in that method is different for each subclass.

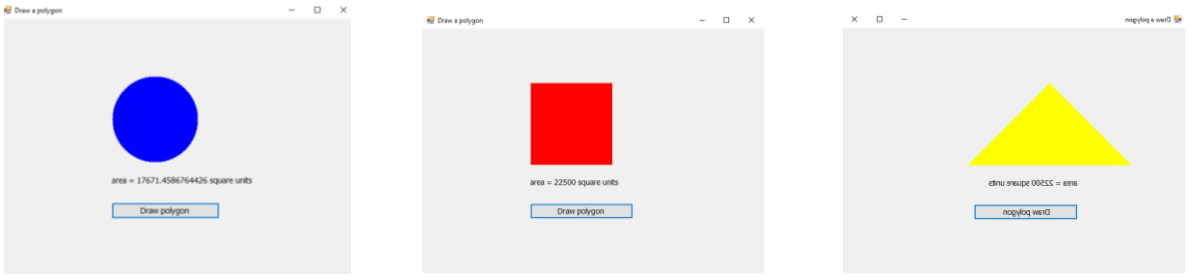
We might write a method that creates a random polygon then displays its area. This method won't know at compile time whether it will create a square, a circle or a triangle. There will therefore be no way for it to know what formula should be used to compute the area. With polymorphism, the method just calls CalculateArea() for whatever shape it has created. The system will dynamically determine what code should be executed by selecting the version of the polymorphic CalculateArea() method that goes with the dynamically created polygon.

Similarly, each of these descendants would have its own implementation for a Draw() method. Squares will use graphics.DrawRectangle and graphics.FillRectangle: circles will use graphics.DrawEllipse and graphics.FillEllipse: and triangles will use graphics.DrawLines (see below for details) and graphics.FillPolygon. The method just says "Draw()" and the object executes its own Draw() handler.

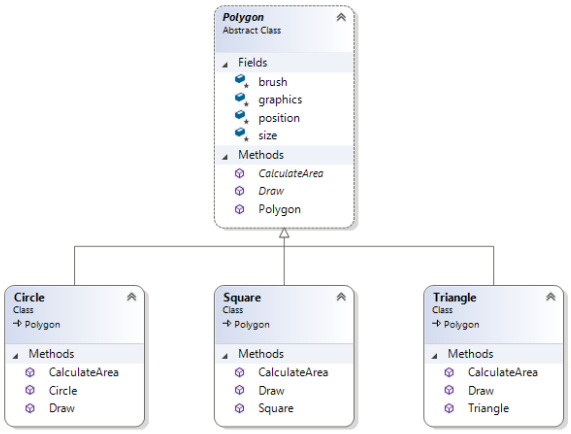
Thus each subclass will **override** the base class' Draw() and CalculateArea() methods.

Task 1: Create a Polygon Class Structure

In this exercise, when the user clicks on a button, a circle, square or triangle is drawn and its area is calculated. The choice of shape is randomly selected on each button1_Click() event.



My UML class diagram is:



The Polygon class is created to provide structure to the hierarchy, but we will never create a Polygon object – we don't have enough information. We will only ever create a Square or Circle or a Triangle. We therefore make the Polygon class **abstract.** This means we declare it, we need it for structure but never implement it by creating an object.

All shapes need to calculate their own area and draw themselves, so these methods are put in the parent Polygon class. However how to calculate the area of a square is different from how we calculate the area of a circle or triangle. So we need a specific version of this method in each of the sub classes. Similarly with the Draw method. This is **polymorphism**. Polymorphism apples **only to methods**, within an inheritance hierarchy.

We need to add some keywords so that the compiler does not get confused by methods with the **same signature** (name, input parameters and return type) in the parent and child classes. In the parent method, we add either the **virtual** (if this method in the parent class will be called) or **abstract** (if this method in the parent class is never called) keyword. In the child class we always add the **override** keyword.

Note, the term **abstract** means **it is never implemented**. This can be used to describe:

- a **class** that is declared but an object of that type is never created, eg the Polygon class.
- a **method** that is declared but never called, eg the Draw() and CalculateArea() methods in the Polygon class.

Abstract classes and abstract methods are shown in **italics** in a UML class diagram.

The position, size, graphics and brush fields are needed for all classes, so these fields are put in the parent class.

1. **Set up the Form1** with a button and a label.

2. Define a **base class Polygon**. Since we will never create an instance of this base class, we declare the class as abstract.

```
public abstract class Polygon
```

3. Polygons should know their x and y positions (these are the upper left corners of the square and circle's bounding rectangle; they are the coordinates of the apex of the triangle, their size and their colour. They need a constructor so that polygon objects can be created. They should also have a Draw method, and a CalculateArea() method.

   In this example, the Draw() and CalculateArea() methods will always be overridden by the subclasses of Polygon. We won't ever use the base class version of Draw() or Calculate Area(), so these methods are declared as abstract.

```
public abstract void Draw();
```

   Note, if the methods could be used by the base class as well as the subclass, the keyword would be virtual rather than abstract.

4. **Descend three classes from Polygon: Square, Circle and Triangle**. These descendants have no new fields, properties or methods, they simply declare their overrides of Draw() and CalculateArea(). For example:

```
public override void Draw()
```

5. Write the code for the base class and its subclasses.

   You can draw the Triangle using the graphics' DrawPolygon() method, which accepts an array of Points, and "connects the dots".

```
graphics.DrawPolygon(pen, new Point[] {position, new Point(position.X + size,
position.Y + size), new Point(position.X - size, position.Y + size) });
```

   Similarly you can colour in the Triangle using the graphics' FillPolygon() method, which accepts an array of Points, and "connects the dots".

```
graphics.FillPolygon(brush, new Point[] {position, new Point(position.X + size,
position.Y + size), new Point(position.X - size, position.Y + size) });
```

6. Create a **Manager class** that has a field called polygon of type Polygon:

```
private Polygon polygon;
```

   Note, you can declare a polygon of type Polygon, and then instantiate it as any of Polygon's subclasses by calling the correct constructor. This is because any object of a subclass is also an exemplar of all its base classes. Just as a cow is a mammal and also an animal, a Triangle is also a Polygon.

   Make a CreatePolygon() method that randomly creates either a Square, a Triangle or a Circle:

```
public void CreatePolygon()
{
     int randomNumber = random.Next(3);
     switch (randomNumber)
     {
       case 0:
          polygon = new Circle(new Point(200, 100), 150, Color.Blue, graphics);
          break;
       case 1:
          polygon = new Square(new Point(200, 100), 150, Color.Red, graphics);
          break;
       case 2:
          polygon = new Triangle(new Point(250, 100), 150, Color.Yellow, graphics);
          break;
        default:
          polygon = null;
          break;
     }
 }
```

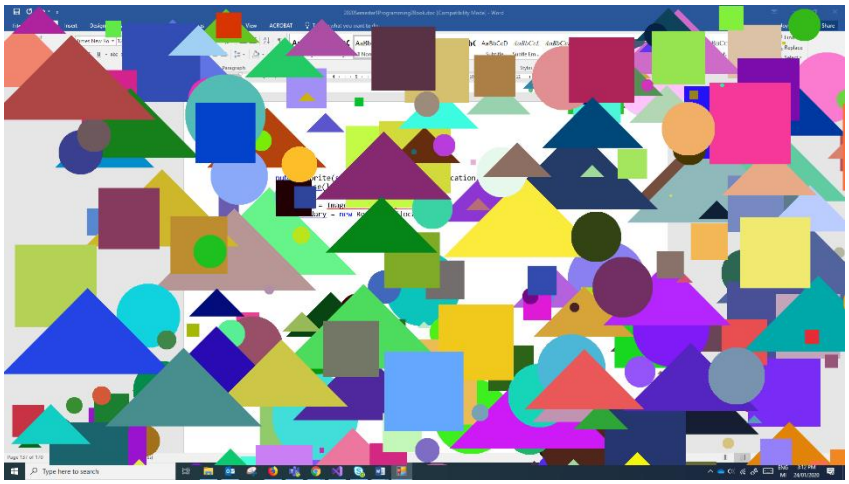
Write a method that tells the chosen polygon to draw itself on the form.

Write a method that tells the polygon to calculate its area and return this value to the form where it will be displayed in the label.

7. Write a `button1_Click` handler for the button so that when it is clicked, it calls the Manager's `CreatePolygon()`, `DrawPolygon()`, `CalculatePolygonArea()` methods for the newly created polygon.

Task 2: Screensaver ******* CHECKPOINT 7 *******

Once you have declared a class structure like Polygon, you can use it in any application where it might be needed. We will build a screen saver that fills the screen with a random assortment of squares, circles and triangles of different sizes and colours. This is my computer screen after the application has been running for about a minute:



To build this application:

1. You need a Timer that, on each tick, generates some Polygons in assorted colours, sizes and locations. In my solution, in the PolygonManager, I generate one Circle, one Square and one Triangle at each timer interval. To generate a nice variety of colours, remember that type Color is not restricted to values of `Color.Blue`, `Color.Red`, etc., but the red, green and blue elements can each be any integer between 0 and the number of colours your screen can display, usually 256. Create your Polygons with their colour property set to

```
Color.FromArgb(random.Next(256), random.Next(256), random.Next(256)).
```

2. When we create a large number of objects and draw them to the screen, we will be consuming a large amount of RAM. To conserve our resources, after we have drawn an object, we should then destroy it, by writing:

```
polygon = null;   // where polygon is the name of the object
```

3. You need to set the Form size equal to the screen size, by setting it's WindowState to Maximised in the Properties window.

4. To make the Form invisible, so that the window behind shows through:

    - Check that the Opacity is set to 100%.
    - Select a colour value for the Form's TransparencyKey from the drop down box. Choose any colour you like. This means that you will be nominating a colour such that any pixels in that colour will be invisible.
    - Set the Form's BackColor property (still in the Properties window) to whatever you selected as the TransparencyKey. This will cause all the pixels of the Form to be invisible; only the window bar at the top of the Form will show.

5. Experiment with the parameters of your application until you get a performance you like. Then modify your `timer1_Tick` handler so that after some number of ticks (enough to let the screen get pretty full), it clears the screen and starts again.

6. One of the advantages of the Object Oriented approach is that the resulting code is easily extensible. Satisfy yourself of this by extending your screensaver to also draw hexagons and pentagons.
   Did you have to modify your Timer handler? If so, how might you have written it so that no modification would be required?
