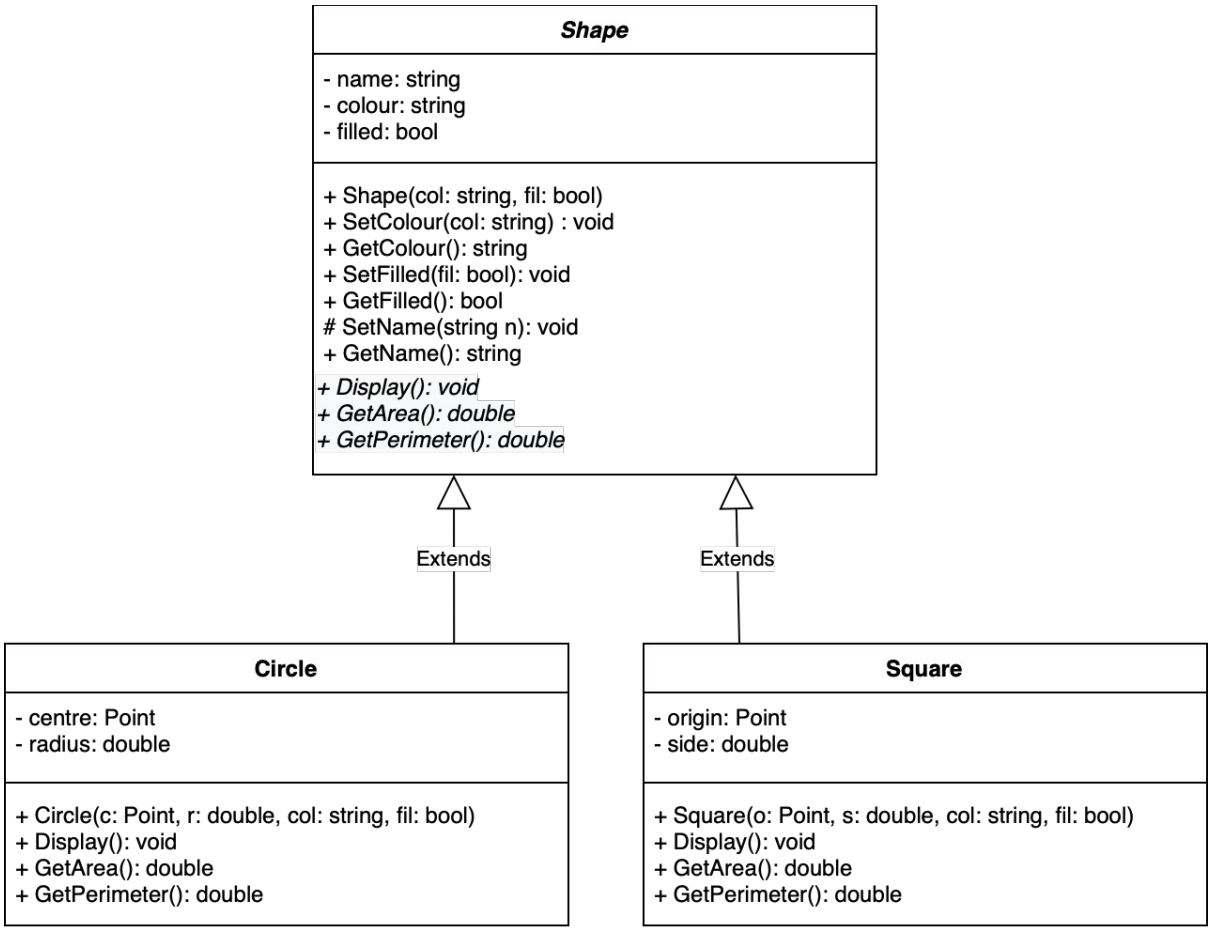
1: Polymorphism, abstract methods and override

During this week's lecture, the OOP principle of Polymorphism was introduced. More specifically, an *abstract* class *Shape* and the subclasses Circle and Rectangle were defined. The following UML Class Diagram complements what was discussed in the lecture by adding more attributes and methods to the abstract *Shape* class:



Note that *protected* members are indicated by the '#' symbol, and *abstract* methods are written using italics font. The *abstract* methods *Display*, *GetArea* and *GetPerimeter* of the *abstract* superclass *Shape* are *overridden* by the subclasses *Circle* and *Square*, as represented in the diagram. More specifically, the body of the *Display* method has to be defined inside each subclass to print the information related to that specific kind of shape (this also includes the shape *colour* and *filled* attributes). Likewise, the body of the abstract *GetArea* and *GetPerimeter* methods will depend on the specific subclass of *Shape*.

Start from the code of the *Circle* and *Point* classes provided below and apply any required changes according to the above Class Diagram. Write the *Square* class so that a square shape has an *origin* (bottom-left vertex, represented via a *Point* object) and a *side* (double). The code of a new *ShapesTest* class is also provided below. Please complete the missing parts and run the program to test your final system implementation.

ShapeTest.cs

```
using System;
namespace Shapes
{
   public class ShapesTest
   {
      static void Main()
      {
         Random r = new Random();

         // missing: declare an array to store the shapes (5 elements)

         string[] colours = { "black", "red", "green", "yellow" };

         for (int i = 0; i < shapes.Length; i++)
         {
            double number = r.NextDouble() * 10; // either the radius or side
            string colour = colours[r.Next(4)];
            int x = r.Next(10);
            int y = r.Next(10);

            bool isFilled;
            if (r.NextDouble() < 0.5)
               isFilled = true;
            else
               isFilled = false;

            if (r.NextDouble() < 0.5)
               // missing: instantiate a Circle using the generated values
            else
               // missing: instantiate a Square using the generated values
         }

          // missing: loop over the array elements and
                  // print the shape name
                  // display the shape information
                  // print the shape area
                  // print the shape perimeter
         // end of the loop
      }
   }
}
```

Point.cs

```
namespace Shapes
{
   class Point
   {
      // attributes that store the information about the point
      // they represent the x and y coordinates
      private int x;
      private int y;

      public Point(int xarg, int yarg)
      {
         x = xarg;
         y = yarg;
      }
```

```
        // returns a string representation of the point
        public override string ToString()
        {
            return $"[{x}, {y}]";
        }
    }
}
```

Circle.cs

```
using System;

namespace Shapes
{
    class Circle
    {
        private Point centre;
        private double radius;

        public Circle(Point c, double r)
        {
            centre = c;
            radius = r;
        }

        public void Display()
        {
            Console.WriteLine("Centre: " + centre.ToString());
            Console.WriteLine("Radius: " + radius);
        }

        public void GetArea()
        {
            Console.WriteLine(Math.PI * radius * radius);
        }

        public void GetPerimeter()
        {
            Console.WriteLine(2 * Math.PI * radius);
        }
    }
}
```

2: Polymorphism, virtual methods and override

In the previous Tutorial (Week 9), we developed a system to represent the information associated with different people who work in a school. You should extend that system this week by introducing an additional *Admin* class. An admin member of staff has a *salary* (double) and a *job title* (string, e.g., “Systems Administrator”, “Payroll employee”, etc.).

Design and develop the new system so that the information related to different object instances can be displayed regardless of their specific type by invoking a method called *Display*. The *Person* class already has such a *Display* method. This should now be declared as a *virtual* method:

```
public void virtual Display()
{
   Console.WriteLine("Name: " + name);
   Console.WriteLine("Surname: " + surname);
   Console.WriteLine("Year of birth: " + yearOfBirth);
   Console.WriteLine("Address: " + address.ToString());
}
```

so that each of the subclasses can *override* it:

```
public override void Display()
{
   // complete with the required code (for each of the subclasses)
}
```

Overridden versions of the *Display* method may reuse the *Display* method already defined inside the *Person* class (hint: use base as discussed during this week's lecture). Also they should print the attributes specific to the type of object on which the method is invoked (e.g., *studentNumber* and *fee* for a *Student*).

Test your code by developing a class that instantiates an object of the *Person* class, as well as one object for each of the subclasses (*Student*, *Teacher* and *Admin*); references to the created object should be stored in an array of *Person* elements, similarly to what was done in the previous exercise. Use a loop to call the *Display* method on the different array's elements to show the polymorphic behaviour of the implemented code.