Faculty of Science, Engineering and Technology

Object Oriented Programming

Pass Task 4.1: Drawing Program — Adding Different Shapes

Overview

Drawing programs have a natural affinity with object oriented design and programming, with easy to see roles and functionality. In this task you will start to create an object oriented drawing program.

Purpose:	See how to use inheritance to create families of related objects and interact with them as a group (polymorphism)
Task:	Extend the shape drawing program to provide multiple different kinds of shapes.
Time:	Aim to complete this task by the start of week 5
Resources:	SplashKit SDK - documentation linked from Canvas

Submission Details

You must submit the following files to Doubtfire:

- Program source code
- Screenshot of program execution

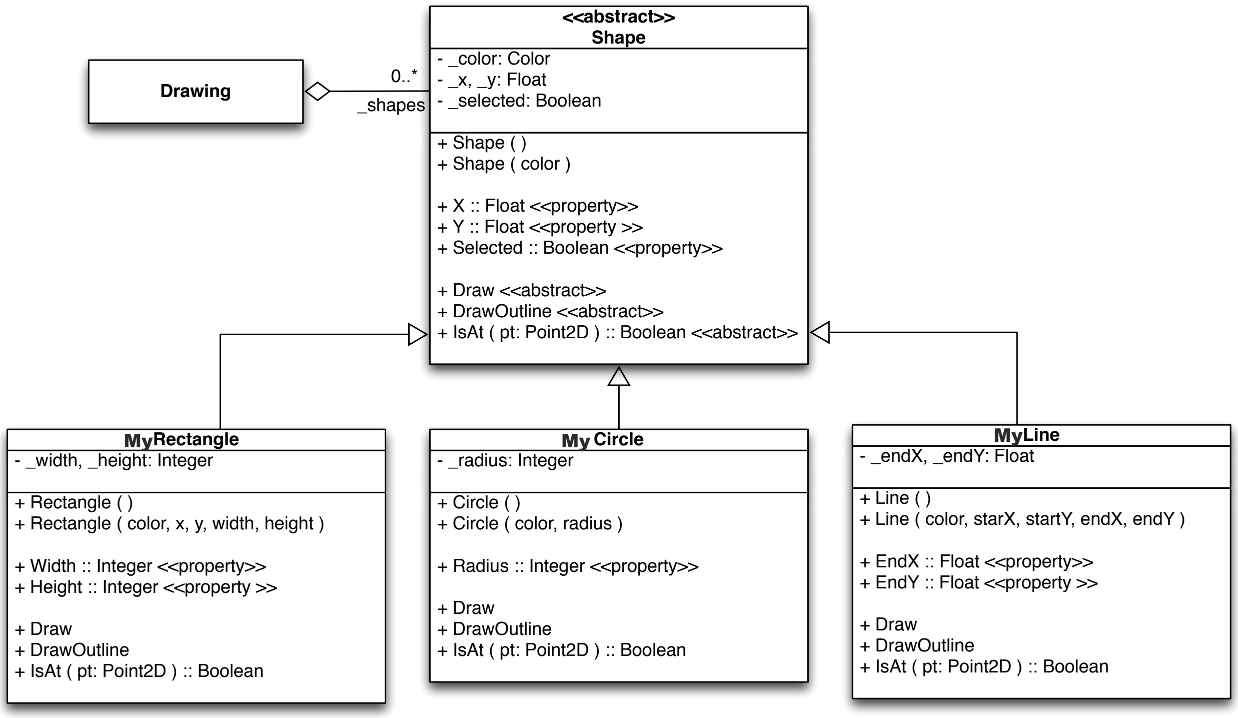
Instructions

This task continues the Shape Drawer from the previous topic. So far you have the ability to create a **Drawing** object that contains many **Shape** objects. Currently, however, all of the shapes are rectangles. What is needed is the ability to use a number of different kinds of shapes.

In the shape drawing program we want to be able to draw a number of different kinds of shapes including rectangles, circles, and lines. One way this could be achieved is with an enumeration and a case statement to determine how methods of the object will behave when different actions are requested. This may be the correct approach in some cases, but there are challenges if you want to store different data associated with the different kinds of objects.

With object oriented programming, **inheritance** can be used to create families of related objects that can be used interchangeably within the program. Different classes can inherit common features from a **parent class** (also know as **base class** or **super class**), and add new features or change how inherited features work. With the Shape Drawing program, we can use this principle to create a family of Shape classes — each of which draws in its own way.

The following UML class diagram shows the inheritance relationships (read as **is-a** or **is-a-kind-of**) that are going to be created in this program. There will be three new classes to represent three different types of shapes we want to support. Each of these classes is a kind of Shape, with each class providing its own custom draw implementation. In a UML class diagram inheritance is drawn as a line with a hollow, triangular, arrow head. This is read **MyRectangle** is a kind of **Shape** etc.



Drawing
0..*
_shapes
<<abstract>>
Shape
- _color: Color
- _x, _y: Float
- _selected: Boolean
+ Shape ( )
+ Shape (color)
+ X :: Float <<property>>
+ Y :: Float <<property >>
+ Selected :: Boolean <<property>>
+ Draw <<abstract>>
+ DrawOutline <<abstract>>
+ IsAt (pt: Point2D) :: Boolean <<abstract>>
MyRectangle
- _width, _height: Integer
+ Rectangle ( )
+ Rectangle (color, x, y, width, height)
+ Width :: Integer <<property>>
+ Height :: Integer <<property >>
+ Draw
+ DrawOutline
+ IsAt (pt: Point2D) :: Boolean
My Circle
- _radius: Integer
+ Circle ( )
+ Circle (color, radius)
+ Radius :: Integer <<property>>
+ Draw
+ DrawOutline
+ IsAt (pt: Point2D) :: Boolean
MyLine
- _endX, _endY: Float
+ Line ( )
+ Line (color, starX, startY, endX, endY)
+ EndX :: Float <<property>>
+ EndY :: Float <<property >>
+ Draw
+ DrawOutline
+ IsAt (pt: Point2D) :: Boolean

Note: Inheritance and polymorphism takes some time to understand, so don't worry if you struggle to understand it at first. As we work through the tutorial we will try to guide your thinking so that you can understand it more fully.

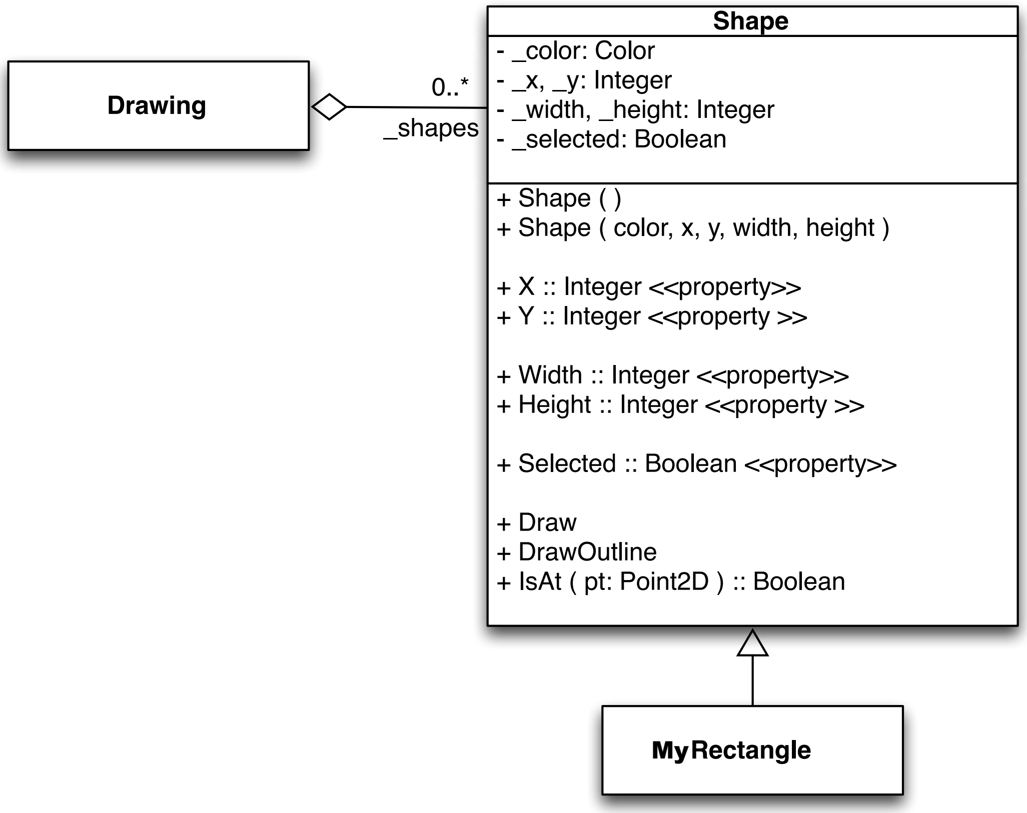
Lets start transforming our program so that it creates the family of Shape types, and uses these in the Drawing.

1. Open your **Shape Drawing** solution.
2. Create a new class called **MyRectangle**.
3. Make **MyRectangle inherit** all of the features from the **Shape** class.

```
public class MyRectangle : Shape
{
}
```

Note: You now have the code setup as shown in the following diagram.

This code gives MyRectangle all of the features of Shape. That means that, at the moment, a MyRectangle knows its location (x,y), its size (width, height), its color, and if it is selected. You can ask a MyRectangle object to Draw, to DrawOutline, and if it IsAt a point. All of these features are **inherited** from the Shape class.

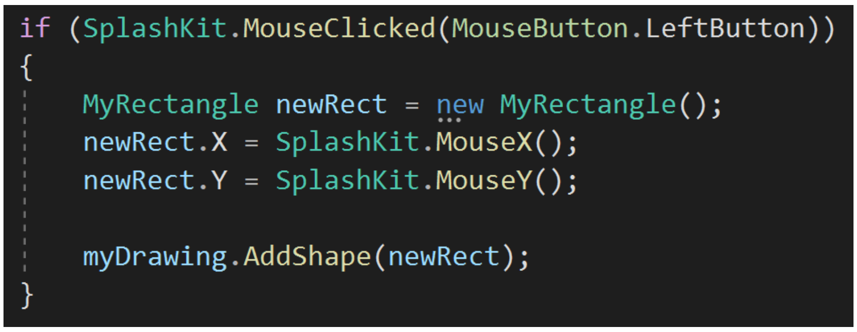

Drawing
0..*
_shapes
Shape
- _color: Color
- _x, _y: Integer
- _width, _height: Integer
- _selected: Boolean
+ Shape ( )
+ Shape (color, x, y, width, height)
+ X :: Integer <<property>>
+ Y :: Integer <<property >>
+ Width :: Integer <<property>>
+ Height :: Integer <<property >>
+ Selected :: Boolean <<property>>
+ Draw
+ DrawOutline
+ IsAt (pt: Point2D) :: Boolean
MyRectangle

Note: What we have looked at so far *is* inheritance, at its core that is all that it is.

Now let us make use of this new MyRectangle. The great thing about **MyRectangle**, in terms of our program, is that it *looks and feels* like a **Shape**. Why, because it **is a kind of** Shape. It can do everything a Shape can — after all, it inherited all of Shape's features, so it *really is* a Shape.

Why is that important? Well, you can **Add** a **Shape** objects to our program's **Drawing** object. So we should be able to add a **MyRectangle** to our Drawing.

4. Switch to **Program** and locate the **Main** method.
5. Change it so that it adds a new **MyRectangle** when the left mouse button is clicked. The code should appear as follows:



```
if (SplashKit.MouseClicked(MouseButton.LeftButton))
{
    MyRectangle newRect = new MyRectangle();
    newRect.X = SplashKit.MouseX();
    newRect.Y = SplashKit.MouseY();

    myDrawing.AddShape(newRect);
}
```

6. Run the program and add some MyRectangles to your Drawing.

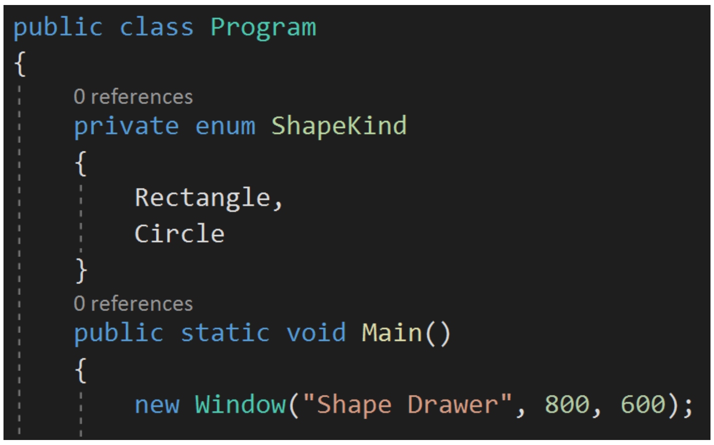
How does this work? This is an example of **subtype polymorphism** - a complex sounding term that basically means you can use things that are Shapes wherever a Shape is required. As a MyRectangle *is a kind* of Shape, you can use it wherever a Shape is required. The Drawing's Add Shape method requires a Shape be passed to it, so you can pass in a MyRectangle.

Note: Polymorphism in general means "many forms", so it relates to any programming aspects where the one *thing* can be used with many different types. In the following list of examples, <u>only subtype polymorphism is specific to object-oriented programming</u>:

- Coercion: `float x = 10;` or `float x = 10.0f;` The assignment can be passed either an integer or a float, integers are converted to become floats.
- Overloading: `FillRect(clr, x, y, w, h);` or `FillRect(clr, myRect);` or … in this case the one *method* has different versions and you can call it in different ways. This is called **overloading**, and is a form of polymorphism.
- Subtype: `Shape s = new MyRectangle();` or `Shape s = new MyCircle();` or … this allows you to use objects of a child class anywhere a parent is required.
- Generics: `List<Shape>` or `List<int>` or … the generic type lets the one List

Now lets try adding MyCircle.

7. Create a new **MyCircle** class, make the class **inherit** from **Shape**.
8. Switch back to **Program** and make the following changes.
    8.1. Declare a ShapeKind within the GameMain class. This will be used by the program to determine what type of shape the user wants to add to the Drawing.



```
public class Program
{
    0 references
    private enum ShapeKind
    {
        Rectangle,
        Circle
    }
    0 references
    public static void Main()
    {
        new Window("Shape Drawer", 800, 600);
```

Note: In C# you can declare types within other types, such as this enumeration within the GameMain class. This type is encapsulated within its enclosing class. This can be useful for simple types, like this enum, where they are only used within the one class and do not really relate to the program overall.

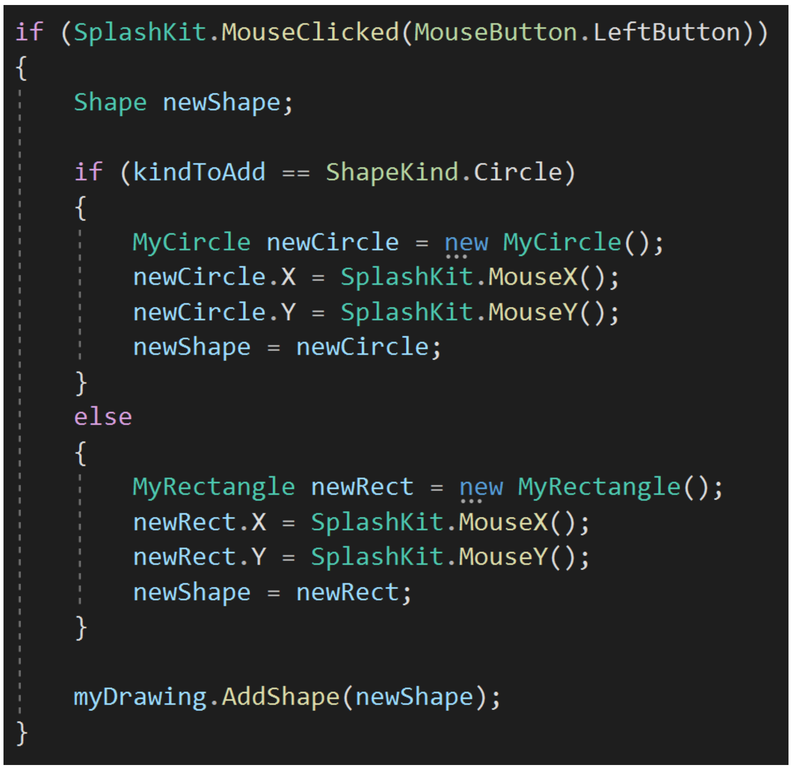
    8.2. Create a ShapeKind variable in **Main** called **kindToAdd**.
    8.3. Initialise kindToAdd with the ShapeKind.Circle value.

8.4. Inside the event loop:

8.4.1. if the user types the **R** key, change the kindToAdd to ShapeKind.Rectangle.

8.4.2. if the user types the **C** key, change the kindToAdd to ShapeKind.Circle.

8.4.3. Change the way the Shape is added so that it creates either a MyRectangle or a MyCircle based on the value in kindToAdd. Use the following code as a guide.



```
if (SplashKit.MouseClicked(MouseButton.LeftButton))
{
    Shape newShape;

    if (kindToAdd == ShapeKind.Circle)
    {
        MyCircle newCircle = new MyCircle();
        newCircle.X = SplashKit.MouseX();
        newCircle.Y = SplashKit.MouseY();
        newShape = newCircle;
    }
    else
    {
        MyRectangle newRect = new MyRectangle();
        newRect.X = SplashKit.MouseX();
        newRect.Y = SplashKit.MouseY();
        newShape = newRect;
    }

    myDrawing.AddShape(newShape);
}
```

Note: See how polymorphism is used to avoid duplicating the call to **Add Shape**. Both MyCircle and MyRectangle objects are kinds of Shape so you can refer to them via a Shape variable.

At the moment, the above code duplicates the code to position the shape. This is un-necessary as **all** Shape objects know their location, so it should be possible to position the Shape independently of which kind of object it is.

9. Correct the duplication of the setting of the shape's location. (See note)

10. Compile and run the program.

When you click and add a shape it will be adding MyCircle objects… but do they really look like Circles? Unfortunately there is no magic to object oriented programs, so even though we called the class MyCircle the computer is still following the same instructions, and so it draws like a MyRectangle. In fact, at this point all Shapes will draw in exactly the same way as the code is all in the Shape class.

We need to rethink how this program is currently designed.

What we want to do is have different Shapes draw in different ways. A MyRectangle object should draw a rectangle, a MyCircle should draw a circle, etc.

With inheritance the child class can add and change behaviour that it inherits from its parent. So we can change how MyCircle works. To get this working we can add a **radius** field that will store MyCircle's radius, and then **override** (change) the Draw and Draw Outline methods so that they actually draw circles.

11. Return to the **MyCircle** class.
12. Add a new **radius** field that stores an integer value, and a **Radius** property to allow others to get and set this value.
13. Add a **constructor** and set MyCircle's radius to 50.
14. Switch to the **Shape** class, and mark both the **Draw** and **DrawOutline** methods as **virtual**. The Draw method is shown below.

```
public virtual void Draw()
{
    if (Selected) DrawOutline();
    SplashKit.FillRectangle(Color, X, Y, Width, Height);
}
```

Note: In C# child classes are only allowed to override methods that are marked as virtual.

15. Switch back to MyCircle and **override** the **Draw** method. It should appear as shown in the following code.

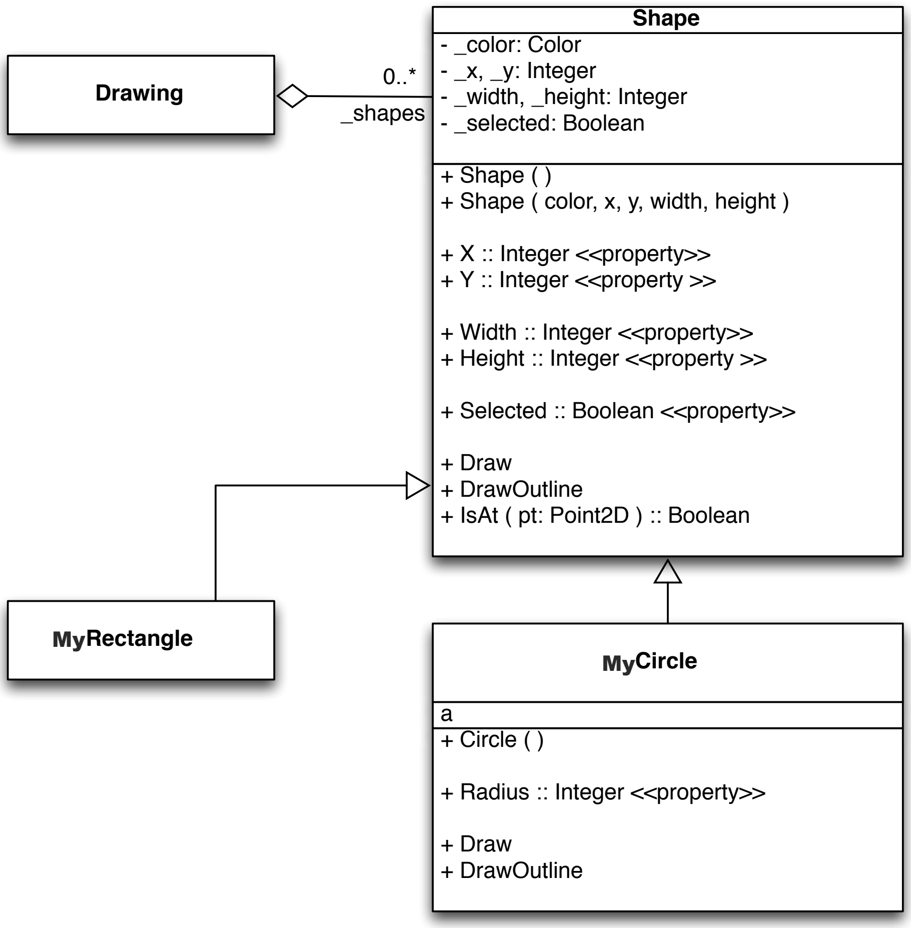
```
public override void Draw()
{
    if (Selected)
        DrawOutline();
    SplashKit.FillCircle(Color, X, Y, _radius);
}
```

Tip: The IDE can help you when overriding methods. Just start typing override and the IDE will give you a list of the methods you can override. By default it calls **base.**Draw() which calls the method from the Shape class.

Note: The fields in the parent class are private, so you will need to use the X and Y properties to access the position for the circle.

16. Run the program. You should now be able to add both Circles and Rectangles, and they should draw correctly.
17. Select a Circle… whoops. Quit the program and return to the **MyCircle** code.
18. Override the **Draw Outline** method and change it so that it now also draws a circle. The radius of the outline should be 2 pixels larger than the Circle's radius.
19. Run the program again, and try selecting and deleting Circles and Rectangles.

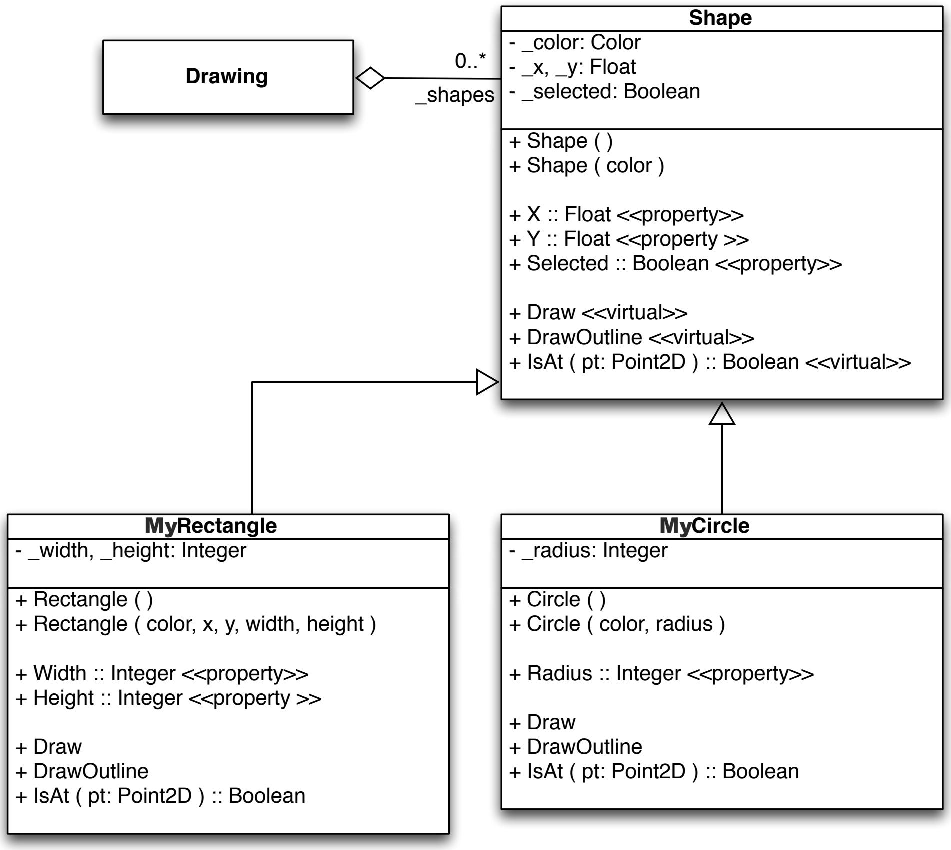
Before we celebrate too much, lets think about what we have created. The following UML class diagram shows the code as it stands. Notice that, at the moment, all Shapes have a Width and Height… but MyCircles also have a Radius. The Width and Height is appropriate for MyRectangle, but not really for the MyCircle, and even less for the MyLine!


Drawing
0..*
_shapes
Shape
- _color: Color
- _x, _y: Integer
- _width, _height: Integer
- _selected: Boolean
+ Shape ( )
+ Shape (color, x, y, width, height)
+ X :: Integer <<property>>
+ Y :: Integer <<property >>
+ Width :: Integer <<property>>
+ Height :: Integer <<property >>
+ Selected :: Boolean <<property>>
+ Draw
+ DrawOutline
+ IsAt (pt: Point2D) :: Boolean
MyRectangle
MyCircle
a
+ Circle ( )
+ Radius :: Integer <<property>>
+ Draw
+ DrawOutline

What we need to do is remove the width and height from the Shape and add it to MyRectangle. The Shape should only contain the logic that is common for ***all*** shapes.

Note: This is going to break some of our unit tests… so it will take a little bit of work. This is why it is best to think about your designs before you start.

20. Rework your Shape, MyRectangle and MyCircle classes so that they match the following UML. The changes are listed below.


Drawing
0..*
_shapes
Shape
- _color: Color
- _x, _y: Float
- _selected: Boolean
+ Shape ( )
+ Shape (color)
+ X :: Float <<property>>
+ Y :: Float <<property >>
+ Selected :: Boolean <<property>>
+ Draw <<virtual>>
+ DrawOutline <<virtual>>
+ IsAt (pt: Point2D) :: Boolean <<virtual>>
MyRectangle
- _width, _height: Integer
+ Rectangle ( )
+ Rectangle (color, x, y, width, height)
+ Width :: Integer <<property>>
+ Height :: Integer <<property >>
+ Draw
+ DrawOutline
+ IsAt (pt: Point2D) :: Boolean
MyCircle
- _radius: Integer
+ Circle ( )
+ Circle (color, radius)
+ Radius :: Integer <<property>>
+ Draw
+ DrawOutline
+ IsAt (pt: Point2D) :: Boolean

- For Shape:
  - Move the width and height fields and properties to the MyRectangle class.
  - Change **Shape constructor** to only accept a Color, have the default constructor call this constructor and pass in Color.Yellow.
  - Change **Draw** and **Draw Outline** to do nothing…
  - Change **Is At** to be virtual and return False.
- For MyRectangle:
  - Create a **MyRectangle constructor** that accepts the color, location (x,y) and size (width,height) for the rectangle. This should call the constructor from Shape that accepts a Color parameter, so it doesn't need to initialise the color itself. See the following code:

```
public MyRectangle (Color clr, float x, float y, int width, int height): base (clr)
{
    X = x;
    Y = y;
    Width = width;
    Height = height;
}
```

    - Have MyRectangle's **default constructor** call the other constructor and pass in Color. Green, 0, 0 for x, y, and 100, 100 for width and height.
    - Override **Draw** and **Draw Outline** to draw a Rectangle.
    - Override **Is At** to check if the point is within the Rectangle.
- For MyCircle:
    - Add a **constructor** that accepts the color and radius, and call this from the default constructor passing in the default color Blue value 50 for radius.
    - Override **Is At** and check if the point is within the circle

21. Compile and run the program. Check that you can add, select, and delete both circles and rectangles.

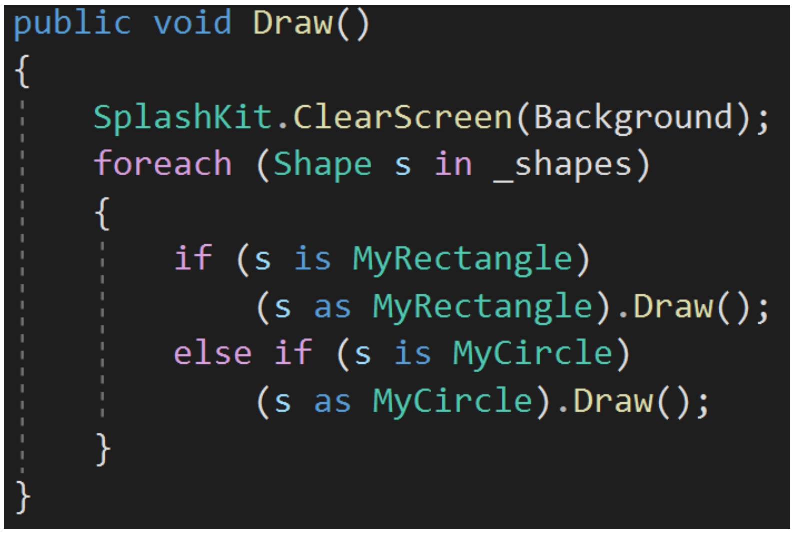
We are now getting close to having this complete. The only problem is that we have some fairly useless methods in Shape… surely there is a better way to implement this.

Currently the **Draw**, **Draw Outline**, and **Is At** methods in Shape really do nothing. They should never be called. However, if you delete them then the program stops working.

22. Give it a try! Delete the **Draw** method from Shape.
23. Build the program to locate the errors.
24. Remove the **override** from both MyRectangle and MyCircle - these are obviously issues. You can't override the method as there is no Draw method in Shape.
25. Build again, and you should find an error with the **Drawing**'s **Draw** method. It was asking its Shapes to **Draw** themselves. Now that Shape does not have a Draw method, the compiler refuses to try to tell the Shape to Draw.

Note: C# checks if it can definitely call methods on objects at compile time. This is called **static typing**. It means that the compiler must be sure it can call the methods at compile time. In contrast, **dynamic typing** allows this to be checked at run time. With dynamic typing the language will try to tell the object to "Draw" (in this case) and if it fails the program will crash. Dynamic typing is more flexible, but static typing is safer.

26. One way to fix this is to **downcast** the object to its explicit type and ask that to Draw. This is not the right approach for this program, but lets see how it works. Change the Draw method in Drawing to match the following code:



```
public void Draw()
{
    SplashKit.ClearScreen(Background);
    foreach (Shape s in _shapes)
    {
        if (s is MyRectangle)
            (s as MyRectangle).Draw();
        else if (s is MyCircle)
            (s as MyCircle).Draw();
    }
}
```

Note: In C# **is** checks if the variable refers to an object of that type. So if s refers to a MyRectangle object, then s is MyRectangle is true. The as operator casts the object to that type, so **s as MyCircle** returns a reference to s' object that knows it is a MyCircle object. If this isn't the case, then s returns **null** - no object.

You can also cast using **((MyCircle) s).Draw();** but this will crash if s does not refer to a MyCircle.

27. Run the program and see that this does work… but in this case there is a better way!

28. Undo your changes to **Drawing**'s **Draw** method, and add **override** back into the method declarations in **MyCircle** and **MyRectangle**.
29. Return to **Shape**.

What we need in Shape is a placeholder that indicates that all Shape objects **must** have a **Draw** method, but that we will *not* provide an implementation here. This can be achieved in C# using the **abstract** keyword.

Note: In C# **abstract** indicates that the child classes of this type **will** override this method and provide its implementation. Anyone can ask a Shape to Draw, but it has no idea how to do this, so it promises its children will know how to do this.

This of this as Shape saying "I promise that any Shape object knows how to draw, I just don't know how they do it!". The language then forces the children to honour this promise.

Once you have an abstract method, there is a gap in the class's code. As a "Shape", you are saying "You can tell me how to Draw… but my children will have the details". So this means, you can no longer create Shape objects. You can create MyRectangles, as they have met this debt, but not Shapes. This requires that you also place an **abstract** marker in the class' definition.

Note: When you have an **abstract class** you can no longer create objects from this class. So it's class object does **not** have a **new** method! No more **new Shape()!**

Note: Do not mix up the idea of abstraction with abstract classes! They both have the word "abstract" in them… but they are distinct concepts. Abstraction is the process by which you come up with ideas for potential classes. Abstract classes are a way of saying "this class does not have everything it needs to create objects" — in effect removing the **new** method from that class.

30. Mark the **Shape** class as **abstract**.

```
public abstract class Shape …
```

31. Mark Shape's **Draw** method as **abstract**.

```
public abstract void Draw(); //thats it!
```

Note: See no implementation… there is nothing there!

32. Build the program, and fix any code where you are calling **new Shape**.

Hint: You should be able to create either a MyRectangle or MyCircle wherever Shapes were created before. Shape s = new MyRectangle(…); for example.

33. Compile and run the program. It should work as before, just now there is no kludgy Draw method implementation in Shape.
34. Change **Draw Outline** and **Is At** to also be **abstract** in **Shape**.

Now for the final piece of the program for this week.

35. Create a **MyLine** class that implements the necessary features to be a kind of Shape.

Hint: Outline by drawing small circles around the start and end points of the line.

36. Adjust Program so that you can switch to add lines when you presses the L key, then click somewhere on the screen.
37. Run the program and check your Line drawing.

Once your program is working correctly you can prepare it for your portfolio. Include your code and screenshots of the working program with circles, rectangles, and lines being drawn.

Assessment Criteria

Make sure that your task has the following in your submission:

- The program must work with Lines, Circles, and Rectangles.
- Code must follow the C# coding convention used in the unit (layout, and use of case).
- The code must compile and the screenshot show it outputting the correct details.
- Should also include some reflections on the need for **abstract** methods and classes.