Faculty of Science, Engineering and Technology

Object Oriented Programming

Topic 4: Class Inheritance and Polymorphism

Resources

The following resources can help you with this topic:

- C# Station Tutorials
  - Class Inheritance
  - Polymorphism
- Tutorials Point C# Programming Tutorials

Topic Tasks

Before starting to work on the tasks, first read through this entire document to get a sense of the direction in which you are heading. Complete the following tasks and submit your work to Doubtfire for feedback before the deadline (check Doubtfire for the submission deadline).

Pass Task 11 - Shape Drawer	2
Pass Task 12 - The Spell Book	15

Remember to submit your progress, even if you haven't finished everything.

After you have **discussed** your work with your tutor and **corrected any issues**, it will be signed off as complete.

Pass Task 11 - Shape Drawer

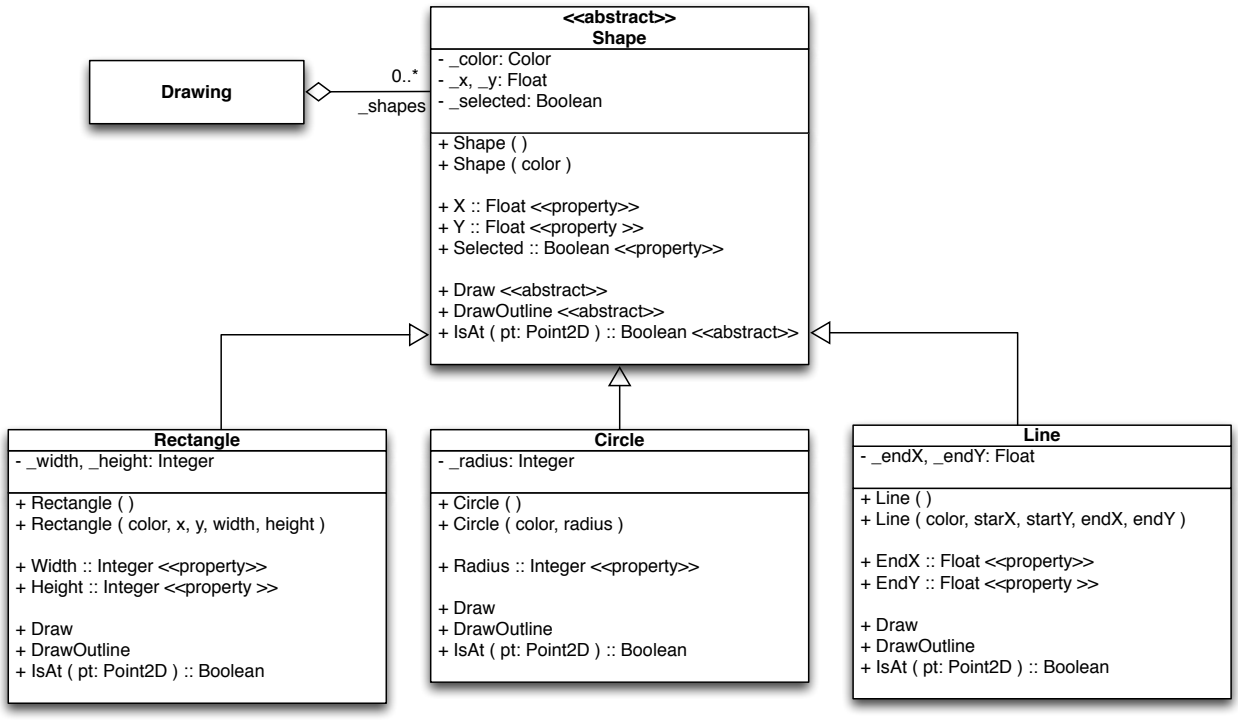
This task continues the Shape Drawer from the previous topic. So far you have the ability to create a **Drawing** object that contains many **Shape** objects. Currently, however, all of the shapes are rectangles. What is needed is the ability to use a number of different kinds of shapes.

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.

> **Note**: The Spell program currently uses an enumeration for Spell Kind, and a case statement to determine how the object behaves.

In 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 objects.

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 **Rectangle** is a kind of **Shape** etc.



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.

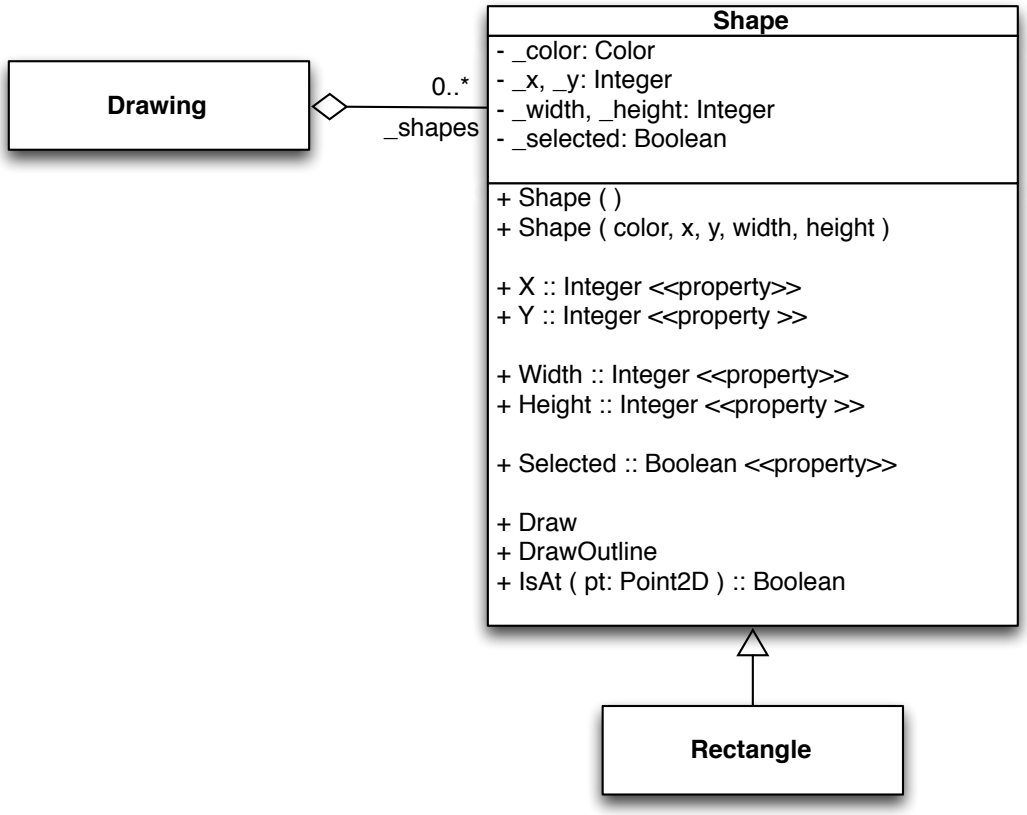
Lets start transforming our program so that it creates the family of Shape types, and uses these in the Drawing.

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

```
public class Rectangle : Shape
{
}
```

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

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



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 Rectangle. The great thing about **Rectangle**, 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 **Rectangle** to our Drawing.

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

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

    myDrawing.AddShape (newRect);
}
```

6. Run the program and add some Rectangles 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 Rectangle *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 Rectangle.

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, only subtype polymorphism is specific to object-oriented programming:

- 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 Rectangle();` or `Shape s = new Circle();` 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 class work as storage for many different kinds of values.

Now lets try adding the Circle.

7. Create a new **Circle** class, make the class **inherit** from **Shape**.
8. Switch back to **Game Main** 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 GameMain
{
    private enum ShapeKind
    {
        Rectangle,
        Circle
    }

    public static void Main()
    {
        //Start the audio system so sound can be played
        SwinGame.OpenAudio();
```

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 Rectangle or a Circle based on the value in kindToAdd. Use the following code as a guide.

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

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

    myDrawing.AddShape (newShape);
}
```

Note: See how polymorphism is used to avoid duplicating the call to **Add Shape**. Both Circle and Rectangle 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 unnecessary 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 Circle objects… but do they really look like Circles? Unfortunately there is no magic to object oriented programs, so even though we called the class Circle the computer is still following the same instructions, and so it draws like a Rectangle. 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 Rectangle object should draw a rectangle, a Circle 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 Circle works. To get this working we can add a **radius** field that will store the Circle's radius, and then **override** (change) the Draw and Draw Outline methods so that they actually draw circles.

11. Return to the **Circle** 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 the Circle'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();
    SwinGame.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 Circle and **override** the **Draw** method. It should appear as shown in the following code.

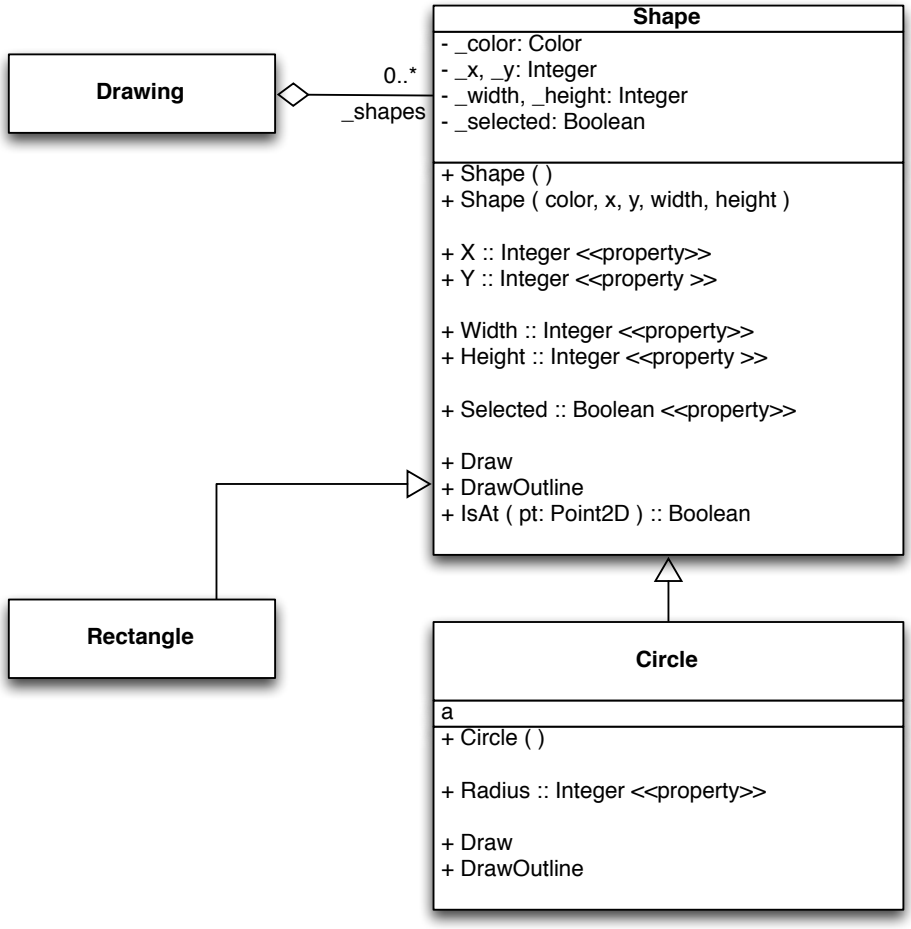
```
public override void Draw ()
{
    if (Selected)
        DrawOutline ();
    SwinGame.FillCircle (Color, X, Y, _radius);
}
```

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 **Circle** 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.

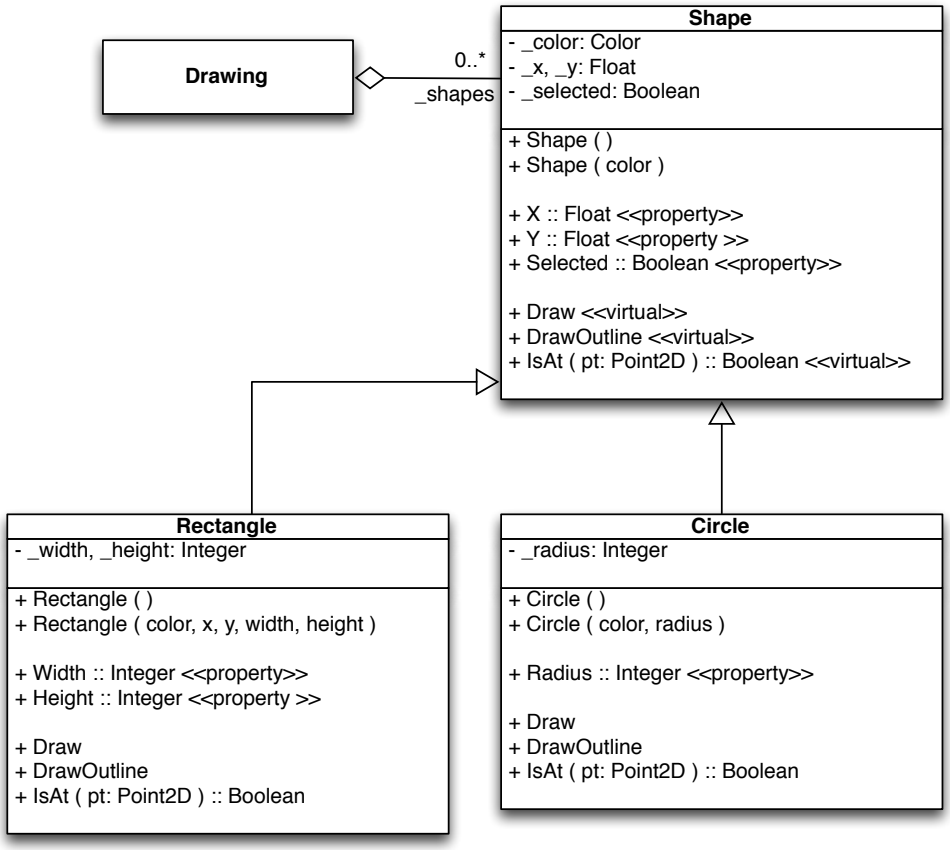
Before we celebrate too much, lets think about what we have created. The following UML class diagram shows the code as it stands. Notice that, at the moment, all Shapes have a Width and Height… but Circles also have a Radius. The Width and Height is appropriate for the Rectangle, but not really for the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



What we need to do is remove the width and height from the Shape and add it to Rectangle. 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, Rectangle and Circle classes so that they match the following UML. The changes are listed below.



- For Shape:
  - Move the width and height fields and properties to the Rectangle class.
  - Change **Shape constructor** to only accept a Color, have the default constructor call this constructor and pass in Color.White.
  - Change **Draw** and **Draw Outline** to do nothing…
  - Change **Is At** to be virtual and return False.
- For Rectangle:
  - Create a **Rectangle 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 Rectangle (Color clr, float x, float y, int width, int height)
    : base(clr)
{
    Color = clr;
    X = x;
    Y = y;
    Width = width;
    Height = height;
}
```

- Have Rectangle'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 Circle:
  - Add a **constructor** that accepts the radius, and call this from the default constructor passing in the default value 50 for radius.
  - Override **Is At** and check if the point is within the circle

At this point your unit test code will not compile due to the creation of Shapes with widths and heights etc. Fix this to get the program working.

21. In the **Drawing Tests** change it to create Rectangle objects but leave the type as an array of Shapes, as shown here:

```
[Test()]
public void TestSelectShape ( )
{
    Drawing myDrawing = new Drawing();
    Shape[] testShapes = {
        new Rectangle(Color.Red, 25, 25, 50, 50),
        new Rectangle(Color.Green, 25, 10, 50, 50),
        new Rectangle(Color.Blue, 10, 25, 50, 50) };
```

22. In the **Shape Tests** change them to use Rectangle variables and objects. For example:

```
[Test()]
public void TestShapeAt ( )
{
    Rectangle s = new Rectangle ();

    s.X = 25;
    s.Y = 25;
```

23. Re-run the tests and make sure they all pass.
24. 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.

25. Give it a try! Delete the **Draw** method from Shape.
26. Build the program to locate the errors.
27. Remove the **override** from both Rectangle and Circle - these are obviously issues. You can't override the method as there is no Draw method in Shape.
28. 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.

29. Get a **screenshot** of this **error** to include in the cover sheet for this program. You can then reflect on this feature of the language.
30. 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 ()
{
    SwinGame.ClearScreen (Background);
    foreach (Shape s in _shapes)
    {
        if ( s is Rectangle )
            (s as Rectangle).Draw ();
        else if ( s is Circle )
            (s as Circle).Draw ();
    }
}
```

> **Note**: In C# **is** checks if the variable refers to an object of that type. So if s refers to a Rectangle object, then s is Rectangle is true. The as operator casts the object to that type, so **s as Circle** returns a reference to s' object that knows it is a Circle object. If this isn't the case, then s returns **null** - no object.
>
> You can also cast using **((Circle) s).Draw();** but this will crash if s does not refer to a Circle.

31. Run the program and see that this does work… but in this case there is a better way!
32. Undo your changes to **Drawing**'s **Draw** method, and add **override** back into the method declarations in **Circle** and **Rectangle**.
33. 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 Rectangles, 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()!**

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

```
public abstract class Shape …
```

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

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

> **Note**: See no implementation… there is nothing there!

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

> **Hint**: You should be able to create either a Rectangle or Circle wherever Shapes were before.

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

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

39. Create a **Line** class that implements the necessary features to be a kind of Shape.

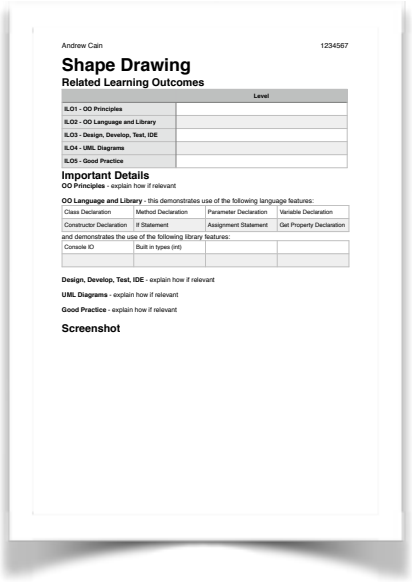
> **Hint**: There is a PointOnLine function, as well as DrawLine functions.

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

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

Once your program is working correctly you can prepare it for your portfolio.

Do not create a new cover page, instead add to your existing Shape Drawing program cover page. Add a few sentences for the learning outcomes you feel that you have demonstrated some aspect of in this work. Include screenshots of the program in action.



Andrew Cain 1234567

Shape Drawing

Related Learning Outcomes

	Level
ILO1 - OO Principles	
ILO2 - OO Language and Library	
ILO3 - Design, Develop, Test, IDE	
ILO4 - UML Diagrams	
ILO5 - Good Practice	

Important Details

OO Principles - explain how if relevant

OO Language and Library - this demonstrates use of the following language features:

Class Declaration	Method Declaration	Parameter Declaration	Variable Declaration
Constructor Declaration	If Statement	Assignment Statement	Get Property Declaration

and demonstrates the use of the following library features:

Console IO	Built in types (int)		

Design, Develop, Test, IDE - explain how if relevant

UML Diagrams - explain how if relevant

Good Practice - explain how if relevant

Screenshot

Pass Task 11 - 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.

Pass Task 12 - The Spell Book

Return to your School of Magic program. This week you will change the implementation to use inheritance to model the different kinds of spells.

1. Use the following description to **draw** your own **UML class diagram** of the Spell Book, Spell class and its child classes.

There are three different kinds of spells: Teleport, Heal, and Invisibility. These output different messages when executed.

- Teleport - these spells only work 50% of the time. When they do work the spell outputs the message "Poof… you appear somewhere else", otherwise it outputs "arrr… I'm too tired to move".

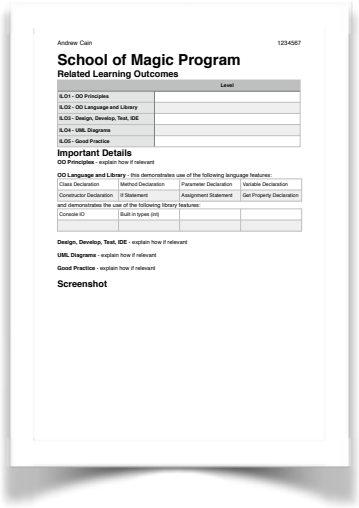
> **Hint**: Create a private **static** Random field in the Teleport spell class, and assign it a new Random object. You can then access this from any Teleport object. To get a random value between 0 and 1 you can use NextDouble.
>
> ```
> private static Random _random = new Random();
>
> …
>
> _random.NextDouble()
> ```

- Heal spells always output the message "Ahhh… you feel better".
- Invisibility spells can only be cast once, after that they output the message "pzzzzit".

> **Hint**: Add a boolean field _wasCast to the Invisibility class. Initialise it to false. You can then check and change this in the Cast method.

2. Add new unit tests to the Spell Test class to check the 3 new derived classes; e.g. test that you can cast an Invisibility spell only once.
3. Implement the Invisibility, Heal and Teleport classes.
4. Modify the Spell Book Test class to check that you can add, remove and fetch spells of any type. (Do you need to modify the Spell Book class itself? Why or why not?)
5. Run your tests and make sure they pass.
6. Add XML documentation to your new classes.

Once your tests are working correctly you can prepare this piece for your portfolio. Do not create a new cover page; instead, add to the existing School of Magic page.



Andrew Cain 1234567

School of Magic Program

Related Learning Outcomes

	Level
ILO1 - OO Principles	
ILO2 - OO Language and Library	
ILO3 - Design, Develop, Test, IDE	
ILO4 - UML Diagrams	
ILO5 - Good Practice	

Important Details

OO Principles - explain how if relevant

OO Language and Library - this demonstrates use of the following language features:

Class Declaration	Method Declaration	Parameter Declaration	Variable Declaration
Constructor Declaration	If Statement	Assignment Statement	Get Property Declaration

and demonstrates the use of the following library features:

Console.IO	Built in types (int)		

Design, Develop, Test, IDE - explain how if relevant

UML Diagrams - explain how if relevant

Good Practice - explain how if relevant

Screenshot

Pass Task 12 - Assessment Criteria

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

- UML class diagram shows the Spell Book, Spell, new classes and their relationships. A photo of a rough sketch is great, as long as it is readable.
- The spell class must be abstract.
- The unit tests correctly check the features of the new spell classes.
- The new classes have XML documentation for all public features.
- Code must follow the C# coding convention used in the unit (layout, and use of case).
- The code must compile and the screenshot show the tests passing.