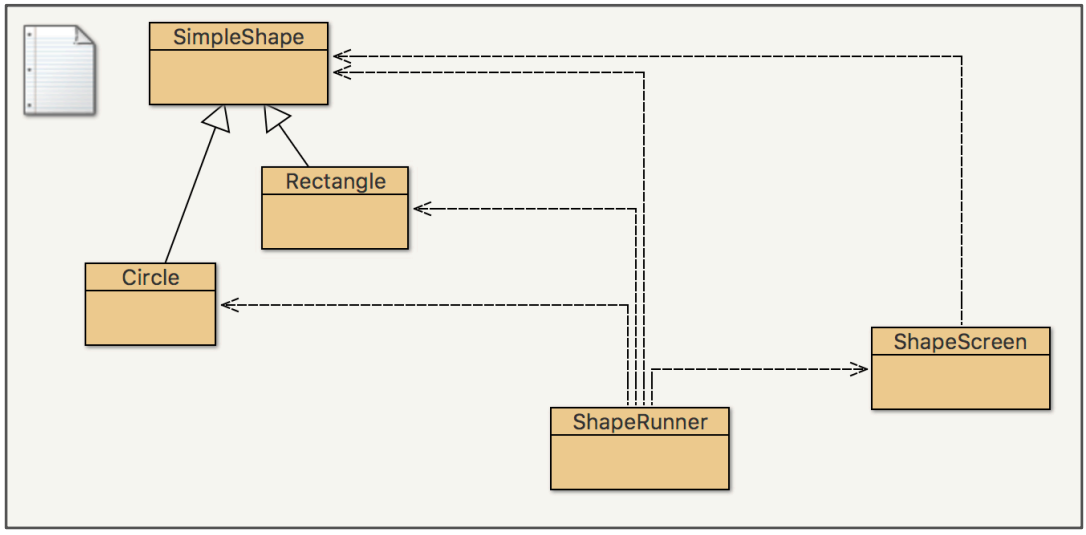
The University of the West Indies, St. Augustine
COMP 2603 Object Oriented Programming 1
2020/2021 Semester 2
Lab 5

In this lab, we will explore the polymorphic behaviour of subclass and superclass instances. This lab builds on the concepts of Inheritance, method overriding and replacement.

Part 1: Polymorphism, Method Binding, Principle of Substitutability



1. Create a new project in BlueJ called Lab 5.
2. Retrieve the following classes from the eLearning course website: **SimpleShape.java, Circle.java, Rectangle.java, ShapeRunner.java** and all 4 of the **.class** files for the **ShapeScreen.** Compile all java files. Execute the **ShapeRunner** file and ensure that the program runs (no output expected).
3. Create the following instances in the **ShapeRunner** class , invoke the **toString( )** method on them and print the output.

Object	Object Type	Features
s1	SimpleShape	
s2	Rectangle	Length = 50, Breadth = 100

TIP: Invoking a parent method from a child class

super.
methodName( )

4. Modify the **toString( )** method (inherited from the **SimpleShape** class) in the **Rectangle** class so that it prefixes the word "Rectangle" to the String produced in the parent **toString( )** method. Execute **ShapeRunner**, and observe the output.

Is this an example of method refinement or method replacement?

5. Change the declaration of the instance **s2** in the **ShapeRunner** class to be of type **SimpleShape**. Observe what happens to the output when you execute the **ShapeRunner** class. Did anything change? Which **toString( )** method was selected for execution on **s2** the one in the parent or the child class? Why?

Answer:

Nothing changed. Because s2 is still a Rectangle object at runtime. So it invokes the Rectangle class's toString() method.

6. Modify the **toString( )** method (inherited from the **SimpleShape** class) in the **Circle** class so that it prefixes the word "Circle " to the String produced by the parent **toString( )** method from SimpleShape. Execute **ShapeRunner**, and observe the output.

7. Create the following instance in the **ShapeRunner** class but declare it to be of type **SimpleShape** and instantiate them as the respective Object type in the table.

TIP: Declaration vs Instantiation

DeclaredClass obj = new InstantiatedClass(..)

Object	Object Type	Features
s3	Circle	Radius = 50

8. Create the following instances in the **ShapeRunner** class but declare and instantiate them as the respective Object type in the table.

Object	Object Type	Features
s4	Circle	Radius = 30
s5	Rectangle	Length = 300, Breadth = 100

9. Invoke the **toString( )** method on the instances from steps 7-8 and print the output. Observe the outcome and identify which **toString( )** method (from the subclass or the superclass) is being called by each instance.

Answer:

Each object invoked their sub-type specific toString() method.

10. Identify the **static** type and the **dynamic** type of each instance in the **ShapeRunner** class.

Answer:
s1: **static: SimpleShape, dynamic: SimpleShape**
s2: **static: SimpleShape, dynamic: Rectangle**
s3: **static: SimpleShape, dynamic: Circle**
s4: **static: Circle, dynamic: Circle**
s5: **static: Rectangle, dynamic: Rectangle**

11. Let's try to reduce the 5 print statements to run in a loop.
    (a) Create an array of 5 **SimpleShape** objects called **shapes**

```
SimpleShape[ ] shapes = new SimpleShape[5];
```

    (b) Insert the 5 objects (**s1..s5**) into the array. Did this work? Why?

```
/* e.g. */   shapes[0] = s1;
```

Answer:

It works! This is because all of the objects are still simple shapes. While some may be rectangles or circles, they still are subclasses of SimpleShape. So this is allowed.

    (c) Type the following code to iterate through the array and print the details of the objects in the array. This is a different way of writing a **for** loop in Java.

```
for (SimpleShape ss: shapes){
    System.out.println(ss.toString());
}
```

    Did this work? What is the **static** type of the objects in the **shapes** array? Why are we able to invoke **toString( )** like this?

Answer: **It works! The static type of the objects in the shapes array is SimpleShape. However, each object could be a SimpleShape, Rectangle or Circle. So we observe that each object's sub-class specific toString() method is invoked.**

12. Override the **calculateArea( )** methods in the **Rectangle** and **Circle** classes so that the **toString( )** method works more correctly.

TIP: Use the Math class in Java to get the value of PI

Math.PI
(import java.lang

13. Invoke the **calculateArea( )** method on the instances within the loop from 11(c). Observe what happens to the output. Why doesn't **s1** have an area? What is the area of a Shape?

Answer: **The output changes for the Rectangle and Circle objects. This is because they override and add functionality to the calculateArea() method. The s1 object is still a SimpleShape variable and that class's calculateArea() method is empty.**

14. Type the following line of code in the **ShapeRunner**:

```
Rectangle s6 = new SimpleShape( );
```

Did this compile? Explain why the compilation error occurs.

Answer:
It does not compile because parent types cannot be converted to sub-types. This is because the parent may not have all of the functionality of the child.

Part 2: Reverse Polymorphism

The **ShapeScreen** class has a method that will render the shapes specified in the array on the Applet window. However, the method requires that all **SimpleShape** objects provide a **draw( )** method that returns a **java.awt.Shape** object.

1. Type the following line of code in the **ShapeRunner**:

```
ShapeScreen screen = new ShapeScreen(shapes); //pass array as param
```

Observe the Applet window displayed. No shapes are displayed. Why not?

Answer: **No shapes are displayed because the draw() method in SimpleShape returns null. Each sub-class of SimpleShape inherits that method be default. This means they will also return null (and not a java.awt.Shape). Therefore, no shapes would be displayed.**

2. How would you override the **draw( )** method in the **Circle** class so that it returns an **Ellipse2D.Double** object with the appropriate dimensions? Examine the constructor of the **Ellipse2D.Double** class, Ellipse2D.Double(double x, double y, double w, double h). It constructs and initialises an **Ellipse2D** object from the specified coordinates.

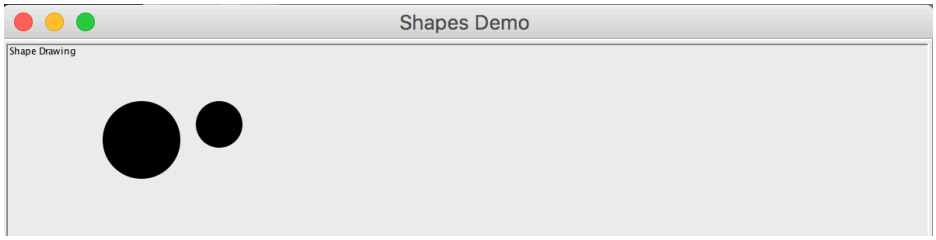
TIP: Visit the API of any Java class to learn more about a method

Google: Java + className

Answer: **We override the draw method in the Circle class by defining a method named draw() that accepts no parameters and returns an Ellipse2D.Double object. This function should not refine its parent's equivalent in any way.**

Override the **draw( )** method in the **Circle** class based on your answer above.

3. Run the **ShapeRunner** class. You should see the following output if your **draw( )** method works properly in the **Circle** class.



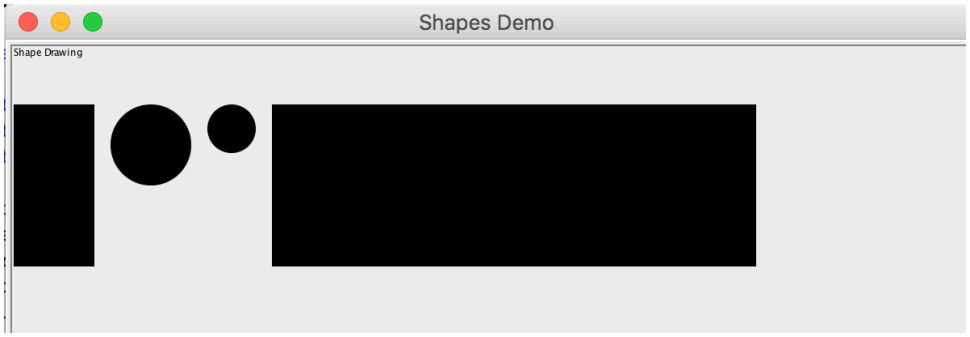
4. How would you override the **draw( )** method in the **Rectangle** class so that it creates and returns a **RoundRectangle2D.Double** object with the appropriate dimensions from the Rectangle class?

   Examine the constructor of the **RoundRectangle2D.Double** class: RoundRectangle2D.Double**(double x, double y, double w, double h, double arcw, double arch) .** It constructs and initialises a **RoundRectangle2D** object from the specified `double` coordinates.

   Answer: **We override the draw method in the Rectangle class by defining a method named draw() that accepts no parameters and returns an RoundRectangle2D.Double object. This function should not refine its parent's equivalent in any way.**

   Override the **draw( )** method in the **Rectangle** class based on your answer above. Set the last two parameters, **arcw** and **arch**, to 0 for now.

5. Run the **ShapeRunner** class. You should see the following output if your **draw( )** method works properly in the **Rectangle** class.

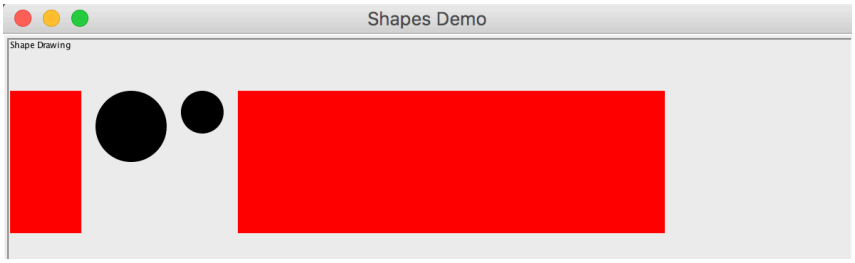


6. Let's try to change the colours of the **SimpleShape** objects.
   - In a **`for`** loop, change the colour of the **SimpleShape** objects to red.
   - Use the mutator to set the colour using Color.red as the parameter.
   - Try some other colours for fun.

TIP: For more colour codes

Google: Java + Color

7. How would you modify your code to generate this colour pattern?



Answer:
We determine which objects are Rectangles in the for loop by using the instanceof keyword. If an object is a Rectangle, then we set its color to red.

TIP: To find out the dynamic type of an object

if (objectname instanceof className)

8. Within your **`for`** loop from Step 6, try to achieve the colour pattern in Step 7 using the **`instanceof`** operator.

   All of the **Rectangle** objects in the pattern are red and all **Circle** objects are black. Why can't we just cast the objects?

Answer:

We can't just cast all of the objects to Rectangles because some of them would not be. This would cause a ClassCastException at runtime, therefore it is unsafe. To cast objects to Rectangles safely, we must first check that they are rectangles using the instanceof keyword.

9. Let's enrich the **Rectangle** class just a bit more.

   Modify the **draw( )** method in the **Rectangle** class such that the last two parameters passed into the constructor of the **RoundRectangle2D.Double** object take the value of the **edgeRoundness** variable. This means that all **Rectangles** will have edges set to the value of **edgeRoundness.** In the **Rectangle** class constructor, the default is 0 which means straight edges.

10. Write a method in the **Rectangle** class called **roundEdge(int curve)** that sets the **edgeRoundness** variable to the incoming value. This would allow us to be change a Rectangle object's edges to rounded.

11. Test your **roundEdge( )** method by invoking it on the instances **s2** and **s5** in the **ShapeRunner** class with a curve of **35**.

    Did it work for both objects? Explain what is happening.

Answer:

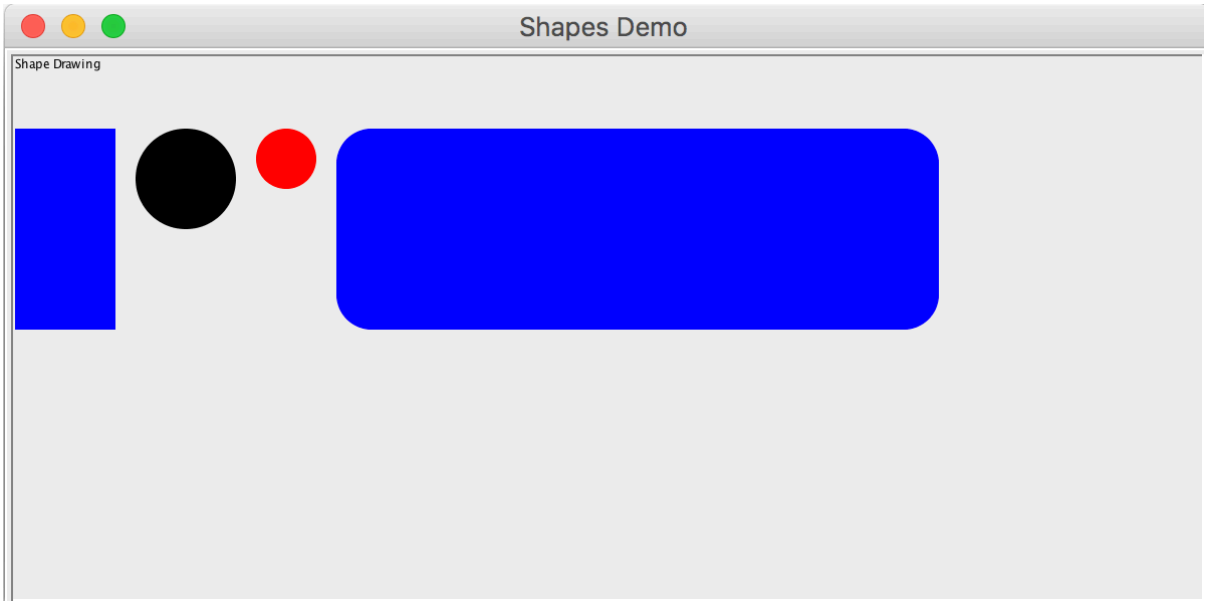
No, it only worked for the s5. This is because s2 has a static type of SimpleShape and SimpleShape does not have a roundEdge() method. In order to invoke this method, we must first cast it back to Rectangle in order to not generate compilation errors.

12. How can you get your **roundEdge( )** method to work on the **Rectangle** objects in the **shapes** array using a loop? Why do you need to cast here?

Answer:

We can cast it back to Rectangle in the for loop if an ss object is an instance of Rectangle. Only then can we access the roundEdge() method.

13. Try to get your code to generate this colour pattern in the **for** loop for the various shapes, and **Rectangle** roundness (curve of 35) :



TIP:

Use getArea( ) and instanceOf