Solutions generated by @markfmyt and AI (subject to being wrong)

Part 1: Polymorphism, Method Binding, Principle of Substitutability Solutions

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?

When you modify the `toString()` method in the `Rectangle` class to prefix the word "Rectangle" to the string produced in the parent `toString()` method, you are **refining the method**. You are adding additional behavior (prefixing the word "Rectangle") and you are also calling `super.toString()` to reuse the behavior of the parent class's `toString()` method.

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?

The reference type is `SimpleShape`, but the object type is `Rectangle`. So, when you call the `toString()` method on `s2`, Java checks at compile time if `toString()` is a method that `SimpleShape` has. It is, because all objects in Java inherit from the `Object` class, which has a `toString()` method.

even though the reference type of `s2` is `SimpleShape`, the `toString()` method of the `Rectangle` class is selected for execution because the actual object type is `Rectangle`. This is an example of polymorphism in Java. The output should be the same as when `s2` was declared of type `Rectangle`. This is because the actual object type, not the reference type, determines which overridden method is used at runtime.

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.

- `s3`: The reference type is `SimpleShape` (superclass), and the actual object type is `Circle` (subclass).
- `s4`: Both the reference type and the actual object type are `Circle` (subclass).
- `s5`: Both the reference type and the actual object type are `Rectangle` (subclass).

In each case, the `toString()` method of the actual object type (subclass) is called due to Java's runtime polymorphism. This means that even if the reference type is the superclass (`SimpleShape`), the subclass's (`Circle` or `Rectangle`) `toString()` method is used because the actual object is an instance of the subclass. This is why we see the output from the `Circle` or `Rectangle` `toString()` method and not the `SimpleShape` `toString()` method. This behavior is a fundamental aspect of polymorphism in object-oriented programming.

So to answer the question, the subclass `toString()` is being called from each instance

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

So far we have:

```
SimpleShape s1 = new SimpleShape();          //inheritence
SimpleShape s2 = new Rectangle (50,100);     //polymorphism
SimpleShape s3 = new Circle(50);             //polymorphism
Circle s4 = new Circle(30);                  //inheritence
Rectangle s5 = new Rectangle(300,100);       //inheritence
```

Static = inheritance

Dynamic = polymorphism

Static type can't be changed, dynamic could be changed is one way to think about it, like we can't say **s4 = new Rectangle(1,1);**

But we can say **s4 = new Circle (25);**

Reason being is that s4 can only be a Circle

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;
```

Yes, inserting the objects `s1` through `s5` into the `shapes` array works. This is because all of these objects are instances of `SimpleShape` or its subclasses (`Circle` and `Rectangle`). In Java, a reference variable of a superclass can be assigned a reference to any object of its subclass, a concept known as upcasting. So, you can assign `s4` and `s5` to elements of the `shapes` array even though they are instances of `Circle` and `Rectangle` respectively.

(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?

invoking `toString()` on each object in the `shapes` array works due to polymorphism. The static type of the objects in the `shapes` array is `SimpleShape`, but the actual object type could be `SimpleShape`, `Circle`, or `Rectangle`. When you call `toString()` on these objects, Java uses the actual object type to determine which `toString()` method to use. So, even if the reference type is `SimpleShape`, the `toString()` method of the actual object type (`Circle` or `Rectangle`) is used. This behavior is a fundamental aspect of polymorphism in object-oriented programming.

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?

The `calculateArea()` method in `SimpleShape` is empty. It does not calculate an area because `SimpleShape` is a generic shape, and we cannot calculate an area without knowing specific details about the shape (like the radius for a circle or the length and width for a rectangle). Therefore, the area of a `SimpleShape` is undefined/empty until it is specified in a subclass like `Circle` or `Rectangle`.

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

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

Did this compile? Explain why the compilation error occurs.

In Java, a `Rectangle` is a `SimpleShape` (since `Rectangle` is a subclass of `SimpleShape`), but a `SimpleShape` is not necessarily a `Rectangle`. It could be any shape, or it could be a `SimpleShape` that is not further specialized at all.
When you try to assign a new `SimpleShape` to a `Rectangle` reference, the Java compiler doesn't allow it because it cannot guarantee that the `SimpleShape` is a `Rectangle`. As per Principle of Substitutability (I think)

Part 2: Reverse Polymorphism Solutions

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?

The `ShapeScreen` class requires that all `SimpleShape` objects provide a `draw()` method that returns a `java.awt.Shape` object. This is because the `ShapeScreen` class uses this method to render the shapes on the Applet window.

So in the `SimpleShape` class, the `draw()` method is defined but it returns `null`:

```
public Shape draw(){
    return null;
}
```

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.

In the `Circle` class, you can override the `draw()` method to return an `Ellipse2D.Double` object. The `Ellipse2D.Double` constructor takes four parameters: the x and y coordinates of the upper-left corner of the framing rectangle, and the width and height of the framing rectangle. For a circle, the width and height are both equal to the diameter, which is twice the radius (but miss have it as 1x the radius):

```
//This is in the circle class and is overriding the superclass' draw
public Shape draw() {
    return new Ellipse2D.Double(x, y, radius, radius);
}
```

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?

```
public Shape draw() {//you can put edgeRoundness in place of 0 (see the lab)
    return new RoundRectangle2D.Double(x, y, length, breadth, 0, 0);
}
```

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?

`I really not sure what miss asking but...`

Casting a `SimpleShape` to a `Circle` or `Rectangle` wouldn't change the fact that it's a `SimpleShape`. It would only allow us to use `Circle` or `Rectangle` methods on that reference.

Furthermore, casting can lead to `ClassCastException` if the object is not actually an instance of the class we're casting to. For example, if `ss` is a `SimpleShape` that's not a `Circle` or `Rectangle`, casting it to `Circle` or `Rectangle` would result in a `ClassCastException`.

So, to safely check the type of an object and perform different actions based on its type, we use `instanceof` rather than/before casting

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.

No it doesn't work for both objects

You're trying to call the `roundEdge(35)` method on `s2` and `s5`. However, `s2` is declared as a `SimpleShape`, and `SimpleShape` does not have a `roundEdge()` method. This is why you're seeing a compilation error when you try to call `s2.roundEdge(35)`.
On the other hand, `s5` is declared as a `Rectangle`, and `Rectangle` does have a `roundEdge()` method. So, `s5.roundEdge(35)` works fine.

We theoritcally put a `roundEdge` in simple shape and have it do nothing and it would compile

To call `roundEdge(35)` on `s2`, you need to downcast `s2` to `Rectangle`. You can do this using a cast, but you should first check if `s2` is actually an instance of `Rectangle` to avoid a `ClassCastException`. Here's how you can do it:

```
if (s2 instanceof Rectangle) {//just cast it to rectangle
    ((Rectangle) s2).roundEdge(35);
}
```

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?

```
for (SimpleShape ss: shapes)
    if (ss instanceof Rectangle)
        ((Rectangle) ss).roundEdge(35);
```

The reason you need to cast here is because `ss` is declared as a `SimpleShape`, and `SimpleShape` does not have a `roundEdge()` method. The `roundEdge()` method is only in the `Rectangle` class. So, you need to tell the Java compiler that `ss` is actually a `Rectangle` so that you can call the `roundEdge()` method. This is done using a cast: `((Rectangle) ss)`. The cast is safe because you've already checked that `ss` is an instance of `Rectangle` with the `instanceof` operator.