COP 3330 – Object Oriented Programming in Java
Interfaces

A class has the following components:

1. Instance Variables
2. Constructors
3. Methods

When Java determines if code is valid to compile, it only needs to know about method signatures. Thus, imagine specifying "less than a class", perhaps a contract that requires certain methods to be written. This is precisely what an interface is.

Once an interface is created, then one can create a class that implements an interface. This is different than inheritance, but can have a similar "feel" to inheritance.

Recall from our inheritance examples that we would pick a base class, such as Coordinate and then we could make an array of Coordinate that could store both Coordinates and ColorCoordinates and naturally process each of the objects accordingly.

For interfaces, we won't "inherit" any code, but we will sign a contract stating that we will implement a set of methods. Once we sign that contract, Java will know if certain method calls are valid or not.

Shape2D Example

For this lecture we'll investigate a single interface: Shape2D. Notice that shapes have too much variety for us to specify any instance variables or constructors for them. But, there are "behaviors" that are common to all two dimensional shapes. Unfortunately, the shapes vary too much for us to write actual code for those behaviors. Thus, all we can do is specify method signatures that correspond to those behaviors that all classes that implement the Shape2D interface have. Here is the code for the full interface, which is stored in Shape2D.java:

```
public interface Shape2D {
    public double getPerimeter();
    public double getArea();
    public void scaleBy(double scaleFactor);
}
```

Literally, we just list method signatures inside of the interface. We can read this as a contract:

If you want to design a class that implements the Shape2D interface, then you must do the following:

1. Write a method called getPerimeter() which returns a double, representing the perimeter of the object of the class.

2. Write a method called getArea() which returns a double, representing the area of the object of the class.

3. Write a method called scaleBy(double scaleFactor) which scales the figure by a linear factor scaleFactor.

If we agree to this contract, that means that I can write the following static method that takes in an array of Shape2D references (so although there's no such thing as an actual Shape2D object since Shape2D isn't a class, we CAN make Shape2D references!!!):

```
public static double sumPerimeter(Shape2D[] shapes) {
    double res = 0;
    for (Shape2D x: shapes)
        res += x.getPerimeter();
    return res;
}
```

Java KNOWS that this will compile because whatever actual objects are referenced by the references in shapes, those corresponding classes are guaranteed to have a getPerimeter() method. At run-time, Java will dynamically bind to the appropriate getPerimeter() method using the rules we previously discussed.

The beauty of this code is that I can have a whole mismash of shapes that each have their own unique ways of computing perimeter and area but I can instantiate all of those different methods via the Shape2D reference without having to separate out my code into a bunch of if statements for every different shape!

So, the benefit of an interface is specifically situations where you know some behaviors in advance (method signatures) that a set of types of objects will adhere to, but you can't pin down any actual instance variables or implementations of those method signatures due to the variations in the set of types of objects. The benefit is that you can write code that takes in references to these all different types of objects that all adhere to the specification and at run-time the decision of which actual method to call will be made based on the actual object types. So, the benefit is similar to that of inheritance without having the full requirements of a base class that true inheritance needs to work.

How to implement an interface

Once we've laid out an interface, and let's face it (pun intended), that task is quite easy, we must now learn how to actually build a class that implements that interface. Recall that we use the keyword "extends" for an inheritance relationship in Java. For an interface, we'll use the key word "implements". Mathematically, the "easiest" class in this example is the rectangle class. Let's just take a look at the whole class and we can dissect the syntax involved (much of which is the same as what we previously learned):

```
public class Rectangle implements Shape2D {

	private double length;
	private double width;

	public Rectangle(double len, double wid) {
		length = len;
		width = wid;
	}

	public double getPerimeter() {
		return 2*(length+width);
	}

	public double getArea() {
		return length*width;
	}

	public void scaleBy(double scaleFactor) {
		length *= scaleFactor;
		width *= scaleFactor;
	}

	public String toString() {
		return "Rectangle "+length+" by "+width;
	}

}
```

The first key takeaway point is that we use the key word implements to indicate that we're implementing an interface with the general syntax of:

```
class CLASSNAME1 implements INTERFACENAME
```

where CLASSNAME1 is the name of the class we're building and INTERFACENAME is the name of the interface that CLASSNAME1 is implementing.

From there, you just write out the code for a regular class with instance variables (and any static variables) listed first, followed by constructors and other methods. The only catch is that you MUST HAVE methods with the identical signatures of those listed in the interface implemented. In this class, we have a constructor (required), the three required methods, as well as a toString method (so that Rectangles don't print out funny).

One can imagine many classes implementing Shape2D. We show Circle next.

Circle implements Shape2D

Here's the Circle class:

```
public class Circle implements Shape2D {

    private double radius;

    public Circle(double myr) {
        radius = myr;
    }

    public double getPerimeter() {
        return 2*Math.PI*radius;
    }

    public double getArea() {
        return Math.PI*radius*radius;
    }

    public void scaleBy(double scaleFactor) {
        radius *= scaleFactor;
    }

    public String toString() {
        return "Circle radius "+radius;
    }
}
```

As you can see, this class has the same methods but just has different implementations.

Triangle implements Shape2D

Finally, let's look at the Triangle class, which is the most complicated. In this implementation, we force users to give us the lengths of two sides of a triangle and the included angle, in between the two sides. (I decided not to allow users to give me three side lengths since it's pretty easy to give three side lengths that don't form a proper triangle.) If the user gives me a negative side length or an invalid included angle (in degrees), I just create an equilateral triangle with side length 1.

Once the three sides of the triangle are set (instance variables), there are some standard geometry formulas that can be used to determine both the perimeter (easy) and area (harder).

Let's look at that implementation on the next page.

```
public class Triangle implements Shape2D {

    private double a;
    private double b;
    private double c;

    public Triangle(double side1, double side2, double includedAngle) {

        if (side1 <= 0 || side2 <= 0 ||
            includedAngle <= 0 || includedAngle >= 180) {

            a = 1;
            b = 1;
            c = 1;
        }
        else {

            a = side1;
            b = side2;
            double radAngle = includedAngle*Math.PI/180.0;
            c = Math.sqrt(a*a+b*b-2*a*b*Math.cos(radAngle));
        }
    }

    public double getPerimeter() {
        return a + b + c;
    }

    public double getArea() {
        double s = getPerimeter()/2;
        return Math.sqrt(s*(s-a)*(s-b)*(s-c));
    }

    public void scaleBy(double scaleFactor) {
        a *= scaleFactor;
        b *= scaleFactor;
        c *= scaleFactor;
    }

    public String toString() {
        return "Triangle with sides "+a+", "+b+" and "+c;
    }
}
```

In the constructor, if the given information is valid, we use the law of cosines to calculate the length of the third side. Since we don't have the height to any base stored in our object, we use Heron's Formula to calculate the area of the triangle. Since this formula uses the semiperimeter, we call the getPerimeter() method in the getArea() method!

Again, this class is pretty much the same as the other two with the corresponding different calculations for each of the methods.

Testing Shape2D

Finally, we'll run some tests on an array of Shape2D references that uses the fact that all Shape2Ds have three required methods written. We'll hard code five shapes (1 Circle, 2 Triangles, 2 Rectangles), print these out, try out a cost method which computes the cost of a service for a set of land that is based on the perimeter and area of the land, and then finally, we'll try out the scaleFactor method. Let's look at the whole program and then dissect it step by step:

```
public class TestShape2D {

    public static void main(String[] args) {

        Shape2D[] shapes = new Shape2D[5];
        shapes[0] = new Circle(5);
        shapes[1] = new Triangle(3, 4, 90);
        shapes[2] = new Triangle(8, 8, 60);
        shapes[3] = new Rectangle(5, 9);
        shapes[4] = new Rectangle(7, 2);

        for (Shape2D x: shapes)
            System.out.println(x);
        System.out.println("\n--------------------------------\n");

        double ans = cost(shapes, 2, 3);
        System.out.println("Cost of these shapes is "+ans);
        System.out.println("\n--------------------------------\n");

        for (int i=0; i<5; i++)
            shapes[i].scaleBy(i+2);

        for (Shape2D x: shapes)
            System.out.println(x+" P = "+x.getPerimeter()+" "+x.getArea());
    }

    public static double sumPerimeter(Shape2D[] shapes) {
        double res = 0;
        for (Shape2D x: shapes)
            res += x.getPerimeter();
        return res;
    }

    public static double sumArea(Shape2D[] shapes) {
        double res = 0;
        for (Shape2D x: shapes)
            res += x.getArea();
        return res;
    }

    public static double cost(Shape2D[] shapes, double costPerLength,
                                                double costPerArea) {
      return sumPerimeter(shapes)*costPerLength + sumArea(shapes)*costPerArea;
    }
}
```

The first few lines creates an array of Shape2D references and fills these with various objects. Since Shape2D isn't a class, we can't actually make any Shape2D objects. Instead, we must put objects of classes that implement the Shape2D interface. Once we do this, we just print these out so we can see that everything is stored as desired. Now, let's carefully investigate the cost method. Imagine that a company has to pay some fee based on the total area and perimeter of its land where the charge per unit length of perimeter is costPerLength and the charge per unit area is costPerArea. To implement this method, we need to easily be able to find the sum of the perimeters of our shapes and the sum of our areas. We farm out that work to two different static methods. We already saw the sumPerimeter method. Now let's look at sumArea:

```
    public static double sumArea(Shape2D[] shapes) {
        double res = 0;
        for (Shape2D x: shapes)
            res += x.getArea();
        return res;
    }
```

This is reasonably straightforward. Start the accumulator at 0, loop through all of the shapes (this implementation uses the iterator loop but a regular for loop through the valid indexes works just as well), and add into the accumulator each shape's area.

Once we have our two methods sumPerimeter() and sumArea(), we just need to call these methods from cost and multiply their return values by the appropriate per unit cost:

```
public static double cost(Shape2D[] shapes, double costPerLength,
                                            double costPerArea) {

    return sumPerimeter(shapes)*costPerLength + sumArea(shapes)*costPerArea;
}
```

Of course, once you write this method, calling it is quite easy!

At the end of the test, we loop through all of the shapes and scale them by 2, 3, 4, 5, and 6, respectively. Our final print shows that our code indeed worked. Here is the full output of this test:

```
Circle radius 5.0
Triangle with sides 3.0, 4.0 and 5.0
Triangle with sides 8.0, 8.0 and 7.999999999999999
Rectangle 5.0 by 9.0
Rectangle 7.0 by 2.0

--------------------------------

Cost of these shapes is 740.5897408543365

--------------------------------

Circle radius 10.0 P = 62.83185307179586 314.1592653589793
Triangle with sides 9.0, 12.0 and 15.0 P = 36.0 54.0
Triangle with sides 32.0, 32.0 and 31.999999999999996 P = 96.0 443.4050067376327
Rectangle 25.0 by 45.0 P = 140.0 1125.0
Rectangle 42.0 by 12.0 P = 108.0 504.0
```