Interfaces

- An `interface` is a like a list of demands
  - It doesn’t implement any code
  - It lists functions you have to implement
- Example: Shape.java

- Can be implemented
  - Circle.java
  - Square.java

Example Interface

```
public interface Shape {
   public double area();
   public double perimeter();
   public void draw();
}
```

<u>This is completely valid Java Code!!!</u>

This is because this is an interface, not a class!

Example Interface

```
public interface Shape {
   public double area();
   public double perimeter();
   public void draw();
}
```

If you want to implement a shape, you have to *"implement"* a way to

- Calculate area
- Calculate perimeter
- Draw the shape (we're going to cheat on this last one)

Circle (***implementing*** Shape)

(truncated for space)

```
public class Circle implements Shape {
   private double radius;

   public Circle(double radius){
   this.radius = radius;
   }
   @Override
   public double area() {
   // TODO Auto-generated method stub
   return radius * radius * Circle.getPi();
   }
   …
```

Circle (***implem*** Shape)

This says "I am behaving like a Shape"

(truncated for space)

```
public class Circle implements Shape {
   private double radius;

   public Circle(double radius){
   this.radius = radius;
   }
   @Override
   public double area() {
   // TODO Auto-generated method stub
   return radius * radius * Circle.getPi();
   }
   …
```

Circle (***implementing*** Shape)

(truncated for space)

```
public class Circ
   private double

   public Circle(
   this.radius = radi
   }
   @Override
   public double area() {
   // TODO Auto-generated method stub
   return radius * radius * Circle.getPi();
   }
   …
```

This says I am overriding the area method. This is not required, but useful.

@Override keyword

- The @Override keyword says I am overriding a “parent” method
  - Such as a method from an interface
- If you remove this keyword from a method that is overriding, it will still work and override it
  - If you have this keyword on a method NOT being overridden, it will cause an error?
- So why use it?

@Override Keyword

- The reason is that interfaces may change
- If you find that you are overriding a method, and suddenly that is causing an error, it’s because the interface has changed
- You can then
  - Find out why the interface changed and,
  - Either
    - Fix the interface
    - Make your code adhere to the new interface

Why use interfaces?

- Consider Circle and Square. Both adhere to the rules of shapes. This means I can say:

```
Shape circle = new Circle(4);
Shape square = new Square(4);
```

- I don’t have to know HOW square and circle work, just THAT they adhere to my Shape interface.
- This will make more sense when we talk about collections

Using abstraction

```
List<Shape> shapes= new ArrayList<Shape>();
shapes.add(new Circle(4));
shapes.add(new Square(4));
for (Shape s : shapes){
   s.draw();
   System.out.println(s.area());
   System.out.println(s.perimeter());
}
```

This is all valid Java code using our examples today. We can treat all Shapes the same, which gives us more flexibility.