CSSE 220 Day 6

Inheritance and Polymorphism

Unit Testing

Inheritance recap:

- Main reasons for inheritance
  - Organization
  - Code reuse
    - Why not just copy and paste the code?
- The usual implication of inheritance: IS-A
  - If we write **A extends B**, it says that an object of type A IS-A object of type B, and can be used as if it is a B.
  - At the very least, it means that A has the same operations as B (perhaps implemented a little bit differently).
  - What if A doesn't override one of B's methods? Can A remove one of B's methods?

Inheritence details: recap

- class A extends B
  - We say that A is a subclass of B and B is the superclass of A.
  - A class can only have one superclass.
  - If you do not include **extends** in a class's definition, that class extends **Object**.
- A has all of the fields and methods B, plus
  - perhaps some new fields
  - almost always some new or overridden methods.
- If A's constructor explicitly Call's B's constructor.
  - Use **super** as the name of the "constructor call".
  - That call must be the first statement in A's constructor code.

One Other Use of inheritance

- **Extension.**
- The subclass has the same operations and can use some of the same code as its parent class (another name for superclass).
- It is closely related to the parent class, though there may not be a strict IS-A relationship.
- Example:
  - `class Point3D extends Point`

```
public class Point3D extends Point {
   private double z;

   public Point3D(double x, double y, double z){
      super(x, y);
      this.z = z;
   }

   public double getZ() {
      return this.z;
   }

   @Override
   public boolean equals(Object other){
      if (other == null)
         return false;
      if (!(other instanceof Point3D))
            return false;
      return super.equals(other) && this.z == ((Point3D)other).z;
   }


   public double distance (Point3D other){
      // what's going to happen if other is Point2D?
      double distance2D = super.distance(other);
      double zDist = this.z - other.z;
      return Math.sqrt(distance2D*distance2D + zDist*zDist);
   }

   @Override
   public String toString(){
      String string2D = super.toString();
      String renamed = string2D.replace("int", "int3D");
      return renamed.replace("]", String.format(",%.2f]", this.z));
   }

}
```

Visibility Modifiers

- **Public** – Accessible by any other class in any package.
- **Private** – Accessible only within the class.
- **Protected** – Accessible only by classes within the same package and any subclasses in other packages.
  - (For this reason, some choose not to use protected, but use private with accessors)
- Default (No Modifier) – Gives package access: accessible by classes in the same package but not by classes in other packages.
  - **Use sparingly! Will be considered an error, unless good reason given for using it. (I will sometimes omit only to fit stuff on slides)**

Can we refactor …

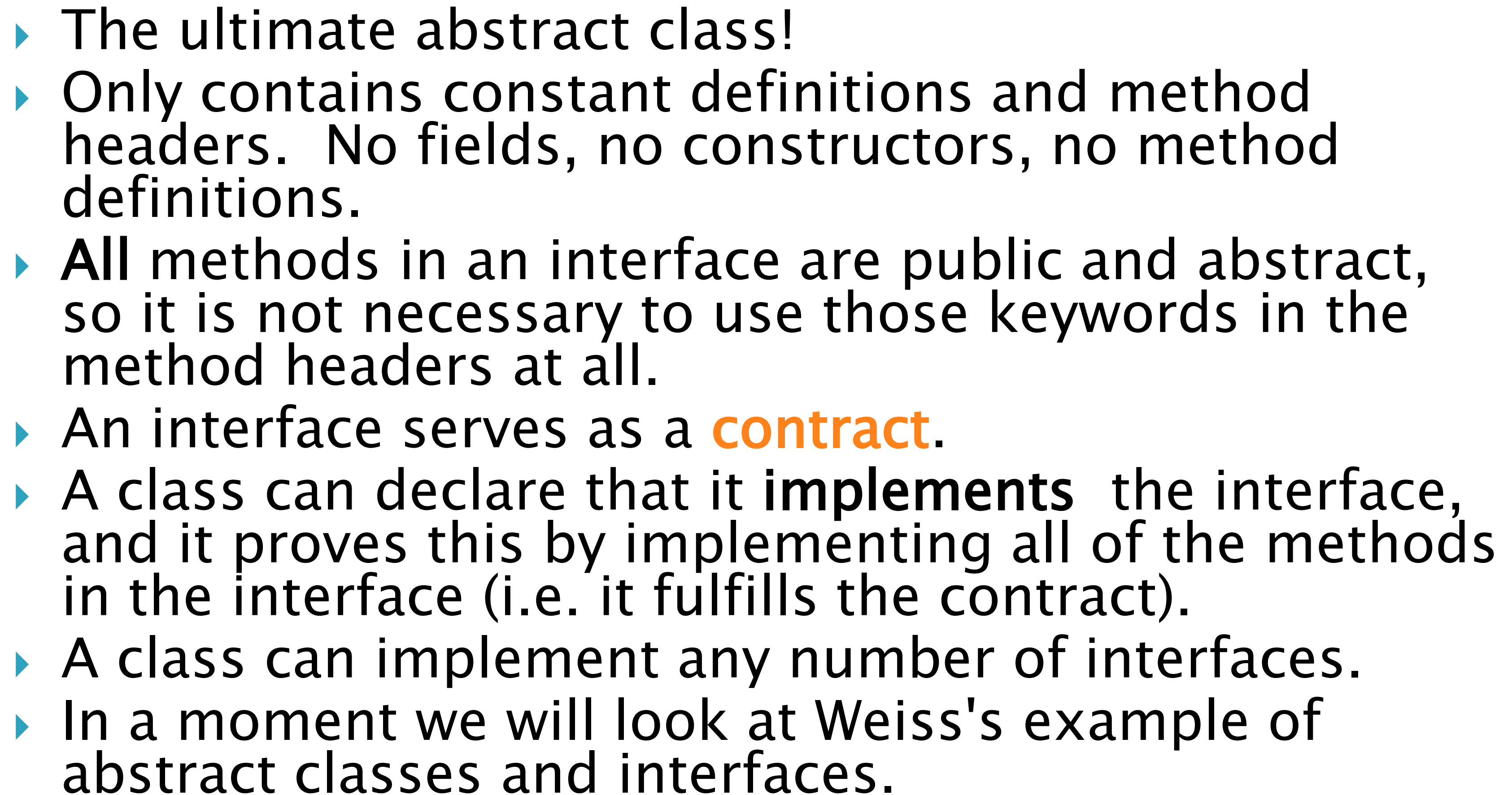
- … to find a common ancestor for Circle and Rectangle?
  - What is a good name for it?
  - What fields/methods can it have?
- We really need an Abstract class. An example soon …

Abstract class

- Gives part of a class definition
  - Intended for other classes to extend it
- Not all methods are defined.
  - For some we just have method headers with a semicolon.
  - Those methods must be declared **abstract**.

- Cannot directly instantiate an abstract class.
- Can instantiate a concrete subclass.
  - The abstract methods must be defined!

Interface



- The ultimate abstract class!
- Only contains constant definitions and method headers. No fields, no constructors, no method definitions.
- **All** methods in an interface are public and abstract, so it is not necessary to use those keywords in the method headers at all.
- An interface serves as a contract.
- A class can declare that it **implements** the interface, and it proves this by implementing all of the methods in the interface (i.e. it fulfills the contract).
- A class can implement any number of interfaces.
- In a moment we will look at Weiss's example of abstract classes and interfaces.

java.util.Comparable interface

- Actually a simplification of Comparable that does not use type parameters
  - We'll discuss type parameters later.

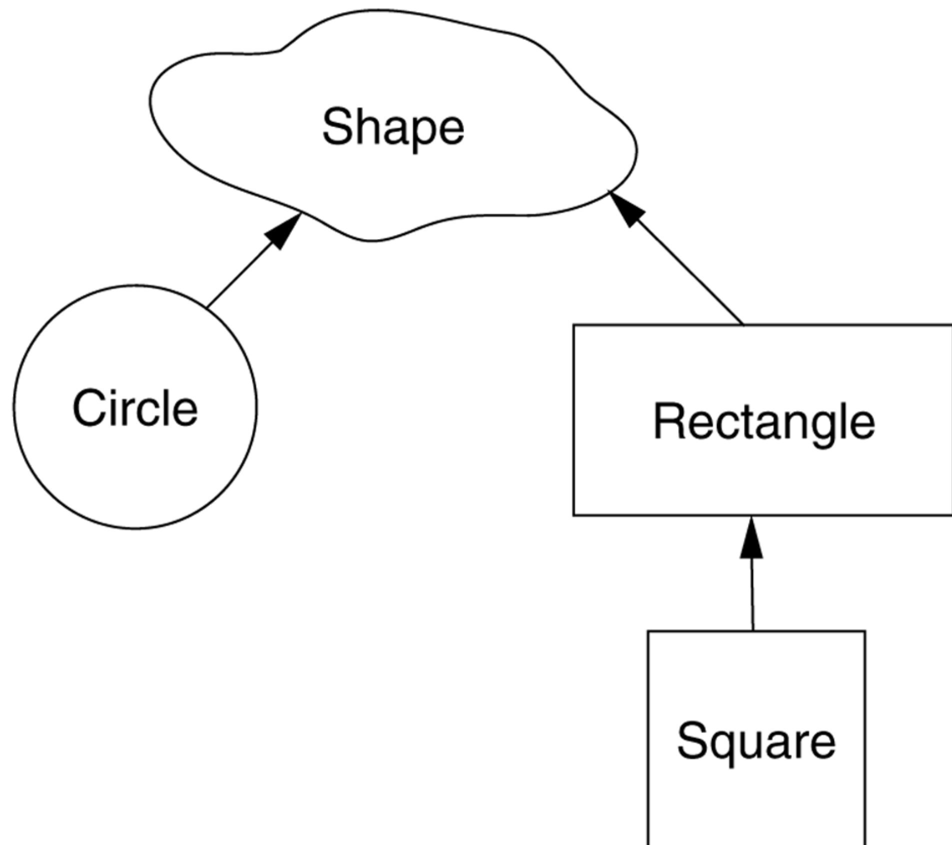
- ```
  public interface Comparable {
     int compareTo(Comparable other);
  }
  ```
  - Returns a positive integer if **this** > **other**, negative if **this** < **other**, zero if **this** ==**other**.

- Any class that says it implements **Comparable** must include the definition of a **compareTo( )** method with the given behavior.
  ```

Shape Hierarchy

Figure 4.10
The hierarchy of shapes used in an inheritance example

Actually, we can (and will) do better, making Shape be an interface, and defining a new abstract class, AbstractShape.



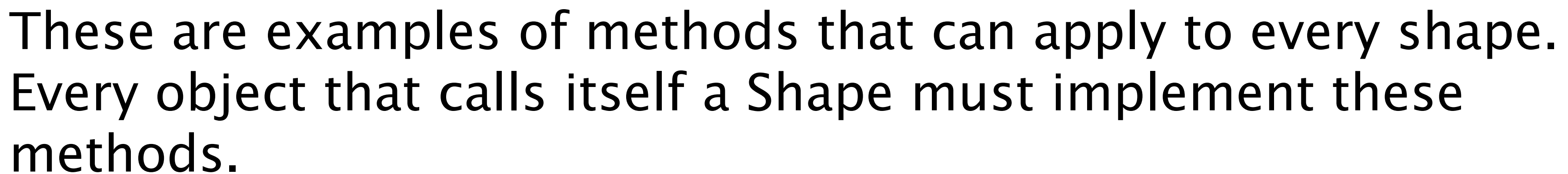
 © 2002 Addison Wesley

Check out the Polymorphism project

- Let’s look at Shape, AbstractShape, Circle, Rectangle, and Square.

The Shape Interface

```java
/* javadoc is omitted in many in-class examples so
code will fit on PowerPoint slides. */

public interface Shape extends Comparable {

    public double area();

    public double perimeter();

    public double semiPerimeter();
}
```



These are examples of methods that can apply to every shape. Every object that calls itself a Shape must implement these methods.

AbstractShape class definition

Note that we can use **area** and **perimeter** in the definitions of **compareTo** and **semiperimeter**, even though the former two methods are not actually implemented in this class.

```
public abstract class AbstractShape implements Shape
{
    public abstract double area( );
    public abstract double perimeter( );

    /* required by the Comparable interface */
    public int compareTo( Object rhs ) {
        double diff = this.area( ) - ((Shape)rhs).area( );
        if( diff == 0 )
            return 0;
        else if( diff < 0 )
            return -1;
        else
            return 1;
    }

    public double semiPerimeter( ) {
        return this.perimeter( ) / 2;
    }
}
```

compareTo is not required to return these specific values (-1 and 1).
Why do you think Weiss does it this way?

Circle class definition

implements a constructor

implements the abstract methods

overrides a method from the Object class

```
public class Circle extends AbstractShape {

    private double radius;

    public Circle( double rad ) {
        this.radius = rad;
    }

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

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

    @Override
    public String toString( ) {
        return "Circle: " + this.radius;
    }
}
```

Rectangle class definition

```
public class Rectangle extends AbstractShape {

        private double length;
        private double width;

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

    public double area( )  {
        return this.length * this.width;
    }

    public double perimeter( ) {
        return 2 * ( this.length + this.width );
    }

    @Override
    public String toString( ) {
        return "Rectangle: " + this.length +
               " " + this.width;
    }

    public double getLength( ) {
        return this.length;
    }

    public double getWidth( ) {
        return this.width;
    }
}
```

implements the abstract methods

overrides a method from the Object class

Methods unique to this class

Square class definition

- Square inherits almost all of its functionality from Rectangle.

```
public class Square extends Rectangle  {
    public Square( double side ) {
        super( side, side );
    }

    public String toString( ) {
        return "Square: " + this.getLength( );
    }
}
```

Interlude

- Sound familiar to anyone?

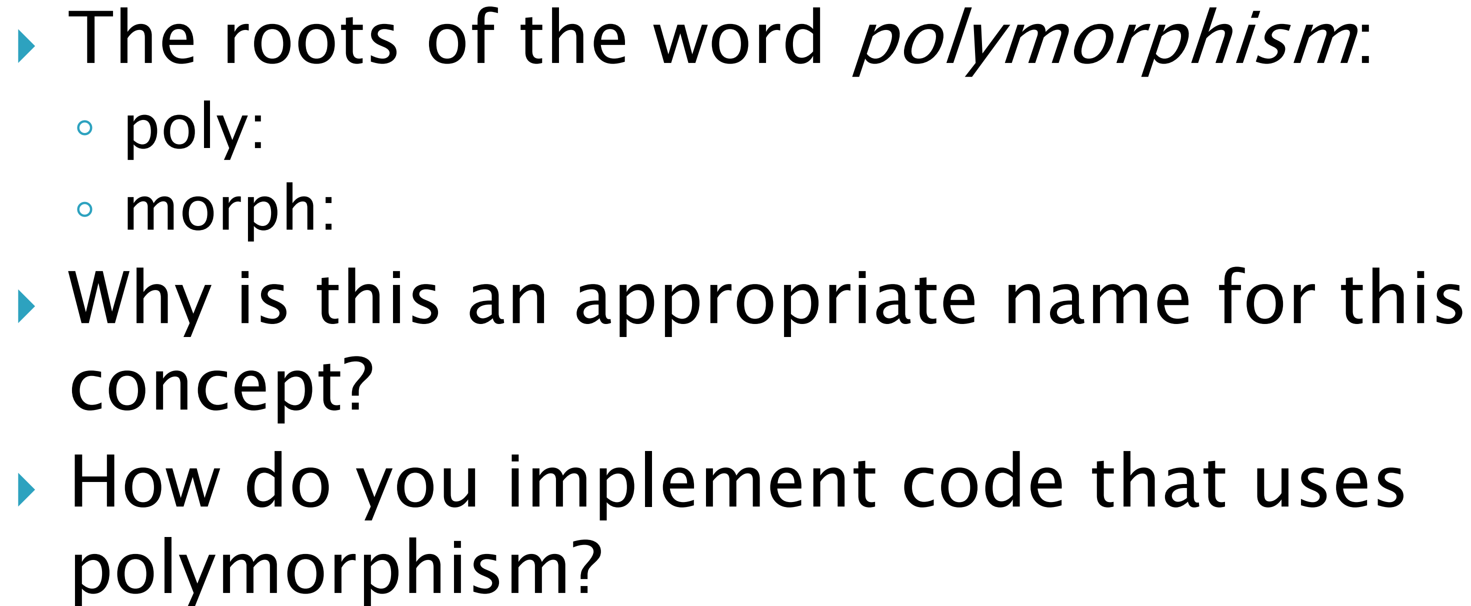
```
public class Workaholic extends Worker {
    public void doWork() {
        super.doWork();
        drinkCoffee();
        super.doWork();
    }


}
```

Interlude

- How about this one?

```
public class RoseStudent extends Worker {
    public void doWork() {
        while (!isCollapsed()) {
            super.doWork();
            drinkCoffee();
        }
        super.doWork();
    }
}
```

Polymorphism

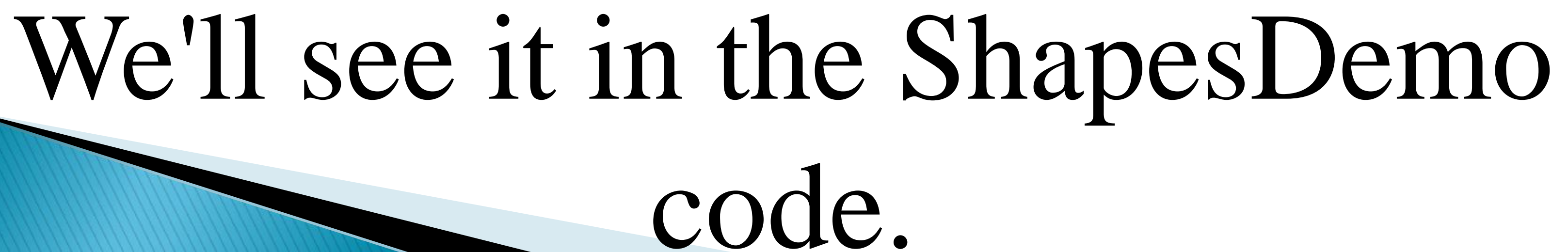


Polymorphism is possible because of ...

dynamic binding of method calls to actual methods.

The class of the actual object is used to determine which class's method to use.

We'll see it in the ShapesDemo code.



Back to the Polymorphism project

- In main(), notice the array of Shapes.
- Please write the missing methods.

Another Example

- In a bird and parrot example, consider a bird method:

```
static void printCall(Bird bird) {
  System.out.println(bird.call);
}
```

```
Bird b = new Parrot();
printBirdCall(b)
Parrot p = new Parrot();
printBirdCall(p)
```

- Generic: printBirdCall expects a Bird, but any type of bird is OK.
- Cannot write Parrot p = new Bird(); //there's not enough info!
- However, without casting, b can only use bird methods.

Casting and `instanceof`

- If we know that b is a Parrot, we can cast it and use Parrot methods (like **speak**):
  ((Parrot)b).speak()

- At runtime, if b is just a Bird, the JVM will throw a ClassCastException.
- To test this, use instanceof:

```
if (b instanceof Parrot) { ((Parrot)b).speak()); }
```

Late Binding: The Power of Polymorphism

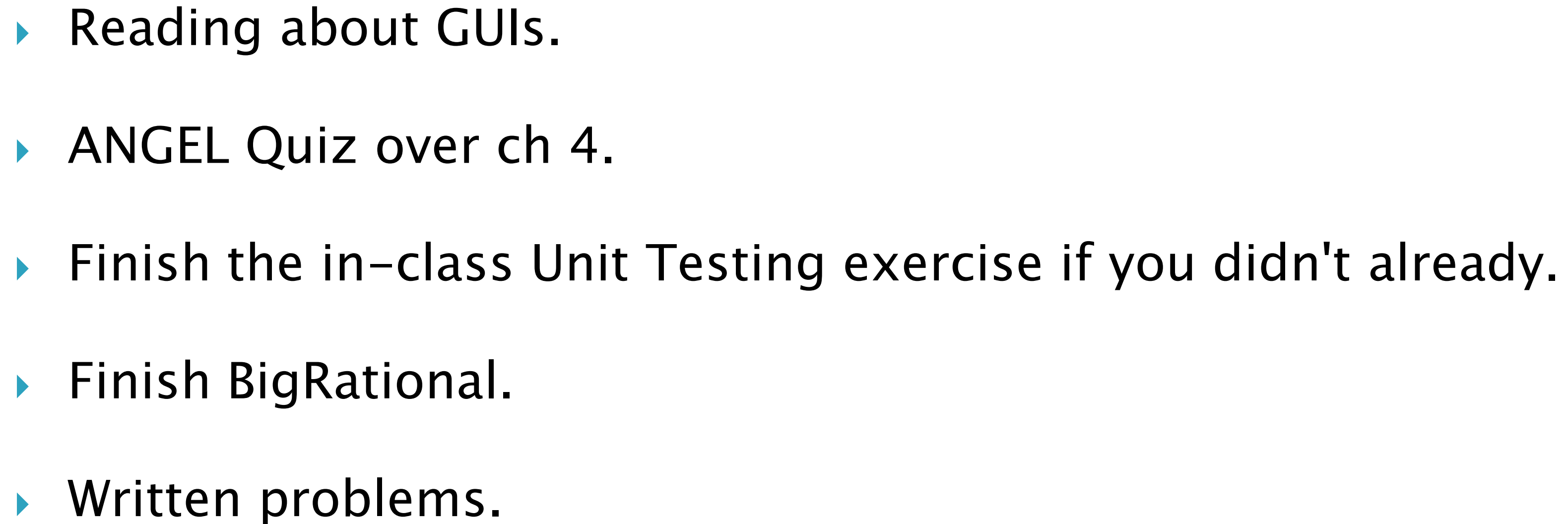
```
Hourly Employee h = new HourlyEmployee("Wilma Worker", new
  Date("October", 16, 2005), 12.50, 170);

SalariedEmployee s = new SalariedEmployee("Mark Manager", new
  Date("June", 4, 2006), 40000);

Employee e = null;
if (getWeekDay().equals("Saturday"))
  e = h;
else
  e = s;
System.out.println(e);
```

When can I tell which value e will have, at **compile time** or **run time**?
So Java defers the decision about which version of toString() will be used until then: it **binds** the actual method call used as **late** as possible.
Late Binding is also called **dynamic dispatch** or **dynamic binding**.
Note: it uses the **most specific version** of the method it can.

Unit Testing and JUnit

- How much testing to do?
  - "Test until fear turns to boredom" – JUnit FAQ.
- **JUnit** is a collection of Java classes that makes it easier to build and run unit tests
- Do the Unit Testing Exercise, linked from the schedule page
- Finish for Homework if you do not finish here.
- If you do finish this early, work on BigRational.

To do before Session 7



- Reading about GUIs.
- ANGEL Quiz over ch 4.
- Finish the in-class Unit Testing exercise if you didn't already.
- Finish BigRational.
- Written problems.