CSSE 220 Day 6

Inheritance and Polymorphism
Unit Testing

CSSE 220 Day 6

- The MineSweeper project will be done by pairs of students. Think about who you'd like to work with. I will ask for your preferences next week or so.
- Some "reality check" changes to my approach:
  - Giving up trying to do solo everything that 3 people did last term. Results:
    - Sometimes no daily quizzes or ANGEL quizzes.
      I'll do them when I reasonably can, and not when I can't.
      Matt can add more next term.
    - Email at night and co-dependency.

Your questions about …

- Java
- Reading from the textbook
- Homework
- etc.

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).

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`

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.

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**.

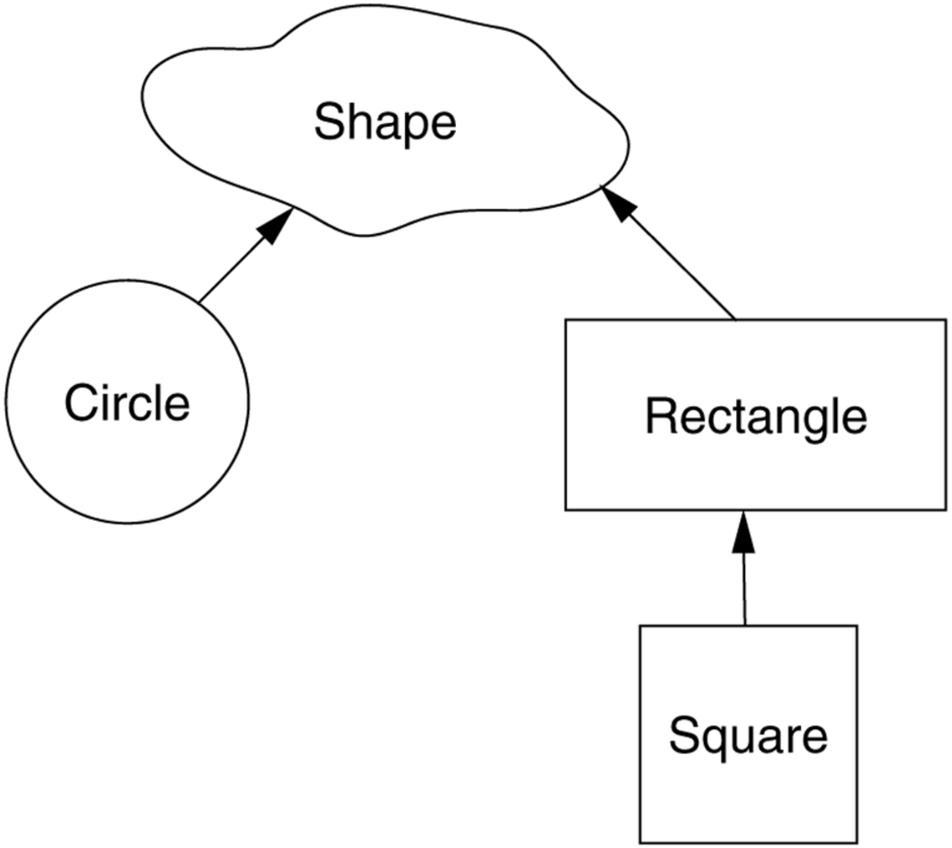
- Any class that says it implements **Comparable** must include the definition of a **compareTo( )** method with the given behavior.

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.



The Shape Interface

```
/* 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

```
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;
   }
}
```

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

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( );
    }
}
```

Polymorphism

- The roots of the word *polymorphism*:
  - poly:
  - morph:
- Why is this an appropriate name for this concept?
- How do you implement code that uses polymorphism?

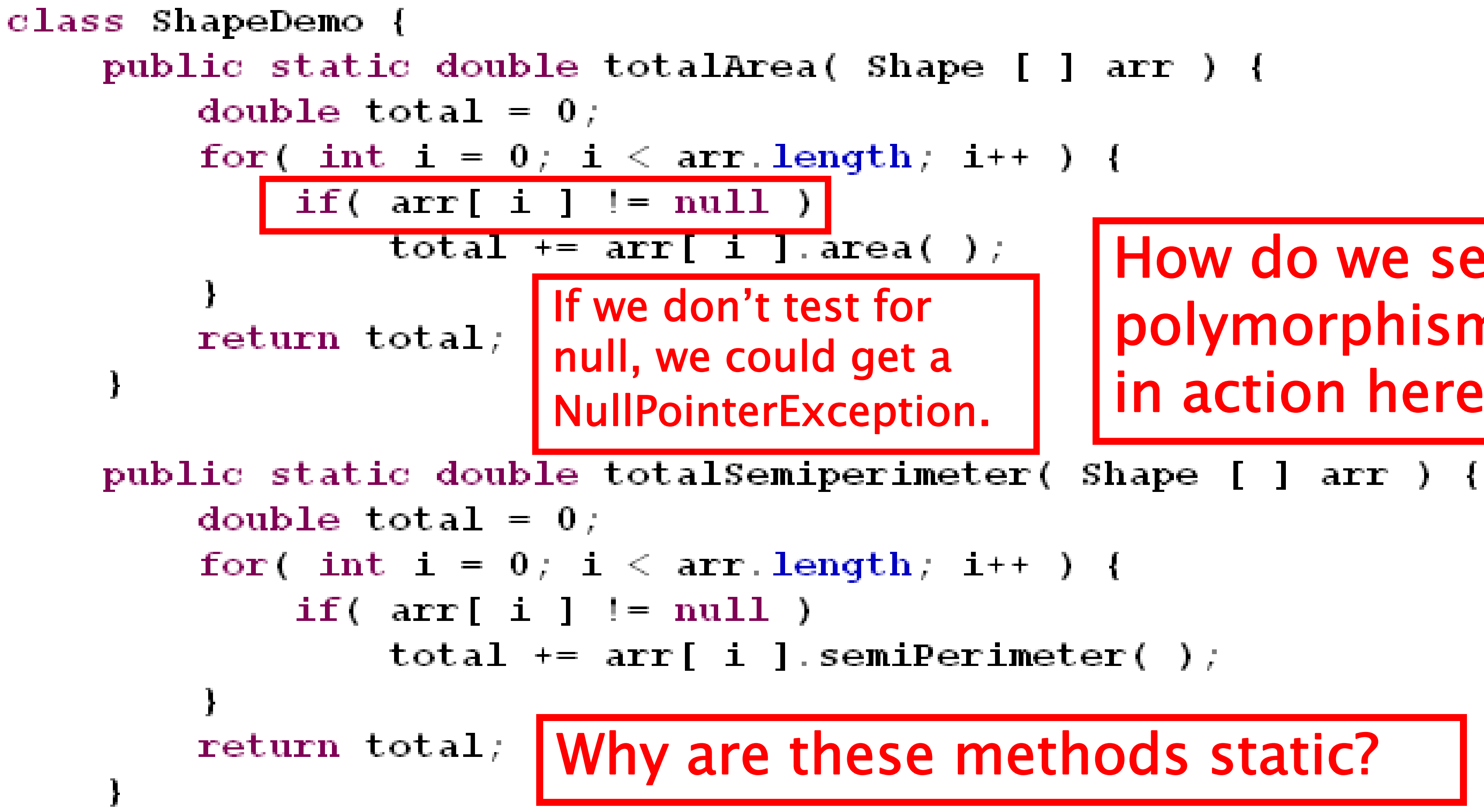
Polymorphism is possible because of …

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.

Shape demo part 1



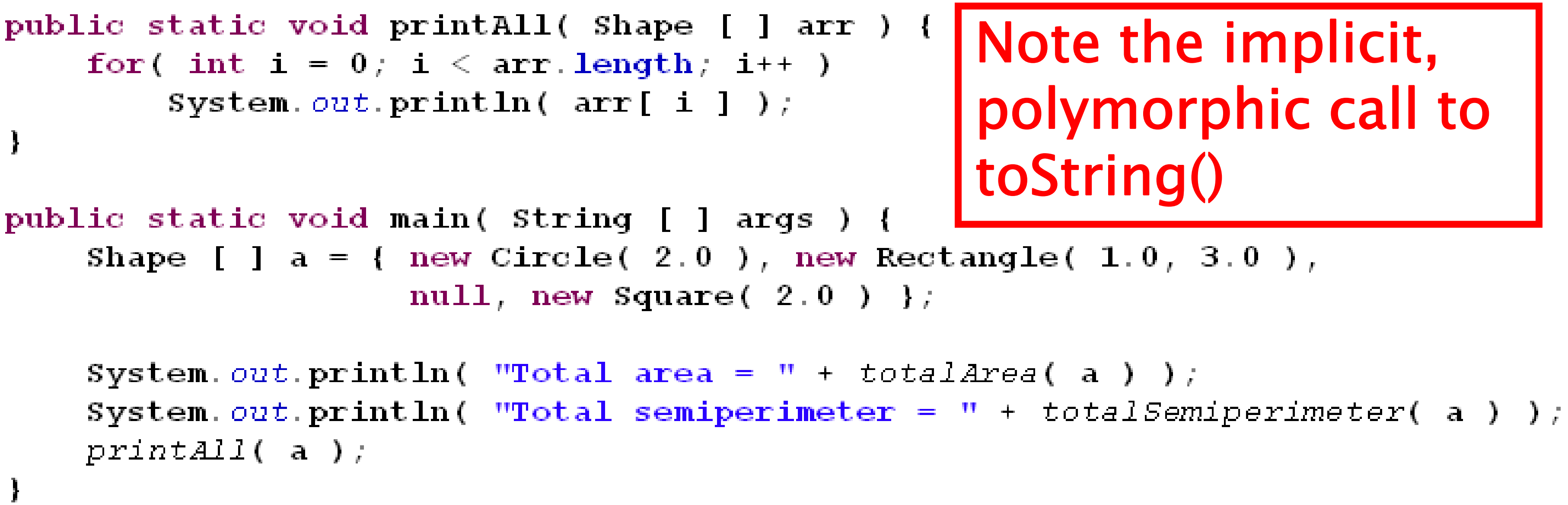
```
class ShapeDemo {
    public static double totalArea( Shape [ ] arr ) {
        double total = 0;
        for( int i = 0; i < arr.length; i++ ) {
            if( arr[ i ] != null )
                total += arr[ i ].area( );
        }
        return total;
    }

    public static double totalSemiperimeter( Shape [ ] arr ) {
        double total = 0;
        for( int i = 0; i < arr.length; i++ ) {
            if( arr[ i ] != null )
                total += arr[ i ].semiPerimeter( );
        }
        return total;
    }
```

If we don’t test for null, we could get a NullPointerException.

How do we see polymorphism in action here?

Why are these methods static?

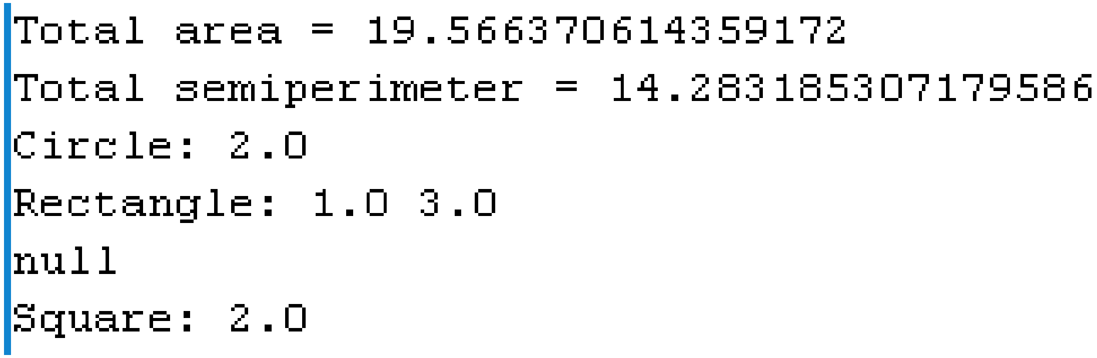
Shape demo part 2



```java
public static void printAll( Shape [ ] arr ) {
    for( int i = 0; i < arr.length; i++ )
        System.out.println( arr[ i ] );
}

public static void main( String [ ] args ) {
    Shape [ ] a = { new Circle( 2.0 ), new Rectangle( 1.0, 3.0 ),
                    null, new Square( 2.0 ) };

    System.out.println( "Total area = " + totalArea( a ) );
    System.out.println( "Total semiperimeter = " + totalSemiperimeter( a ) );
    printAll( a );
}
```

Note the implicit, polymorphic call to toString()

Output:



```
Total area = 19.566370614359172
Total semiperimeter = 14.283185307179586
Circle: 2.0
Rectangle: 1.0 3.0
null
Square: 2.0
```

Interlude

Please do this silently, so you will not spoil it for anyone else.

I will present you with something to look at and a question about it.
You will have about 10 seconds. Again, don't say anything aloud.

Count every "F" in the following text:

**FINISHED FILES ARE THE RE
SULT OF YEARS OF SCIENTI
FIC STUDY COMBINED WITH
THE EXPERIENCE OF YEARS...**

HOW MANY?

Try again

Now that you know what to expect, try again. Do you get the same count? Again, do not say anything.

Count every "F" in the following text:

**FINISHED FILES ARE THE RE
SULT OF YEARS OF SCIENTI
FIC STUDY COMBINED WITH
THE EXPERIENCE OF YEARS...**

HOW MANY?

Third try

Hint: The correct answer is NOT _____

Count every "F" in the following text:

**FINISHED FILES ARE THE RE
SULT OF YEARS OF SCIENTI
FIC STUDY COMBINED WITH
THE EXPERIENCE OF YEARS...**

HOW MANY?

Fourth try

There are actually ____ of them.
Can you see them?

Count every "F" in the following text:

**FINISHED FILES ARE THE RE
SULT OF YEARS OF SCIENTI
FIC STUDY COMBINED WITH
THE EXPERIENCE OF YEARS...**

The Answer

Count every "F" in the following text:

**FINISHED FILES ARE THE RE
SULT OF YEARS OF SCIENTI
FIC STUDY COMBINED WITH
THE EXPERIENCE OF YEARS...**

Unit Testing and JUnit

- **Unit Testing:**
- Test each class/method, independent of the larger program in which they live.
- 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

- The next reading assignment.
- ANGEL Quiz over Section.
- Finish the in-class Unit Testing exercise if you didn't already.
- Finish BigRational.
- A couple more written problems.
- Written problems and ANGEL quiz should be available this afternoon.