CSSE 220 Day 15

Inheritance
Polymorphism
Abstract Classes

Check out *Inheritance* from SVN

Questions?

Inheritance

- Sometimes a new class is a special case of the concept represented by another
- Can "borrow" from an existing class, changing just what we need
- The new class inherits from the existing one:
  - all methods
  - all instance fields
- Can add new fields/methods
- Or override existing methods

Q1

Code Examples

- `class SavingsAccount extends BankAccount`
  - adds interest earning, while keeping other traits
- `class Employee extends Person`
  - adds pay info. and methods, keeps other traits
- `class Manager extends Employee`
  - adds info. about employees managed, changes pay mechanism, keeps other traits

Notation and Terminology

- ```
  class SavingsAccount extends BankAccount {
      // added fields
      // added methods
  }
  ```
- Say "SavingsAccount is a BankAccount"
- Superclass: BankAccount
- Subclass: SavingsAccount

Q2
  ```

Other natural examples

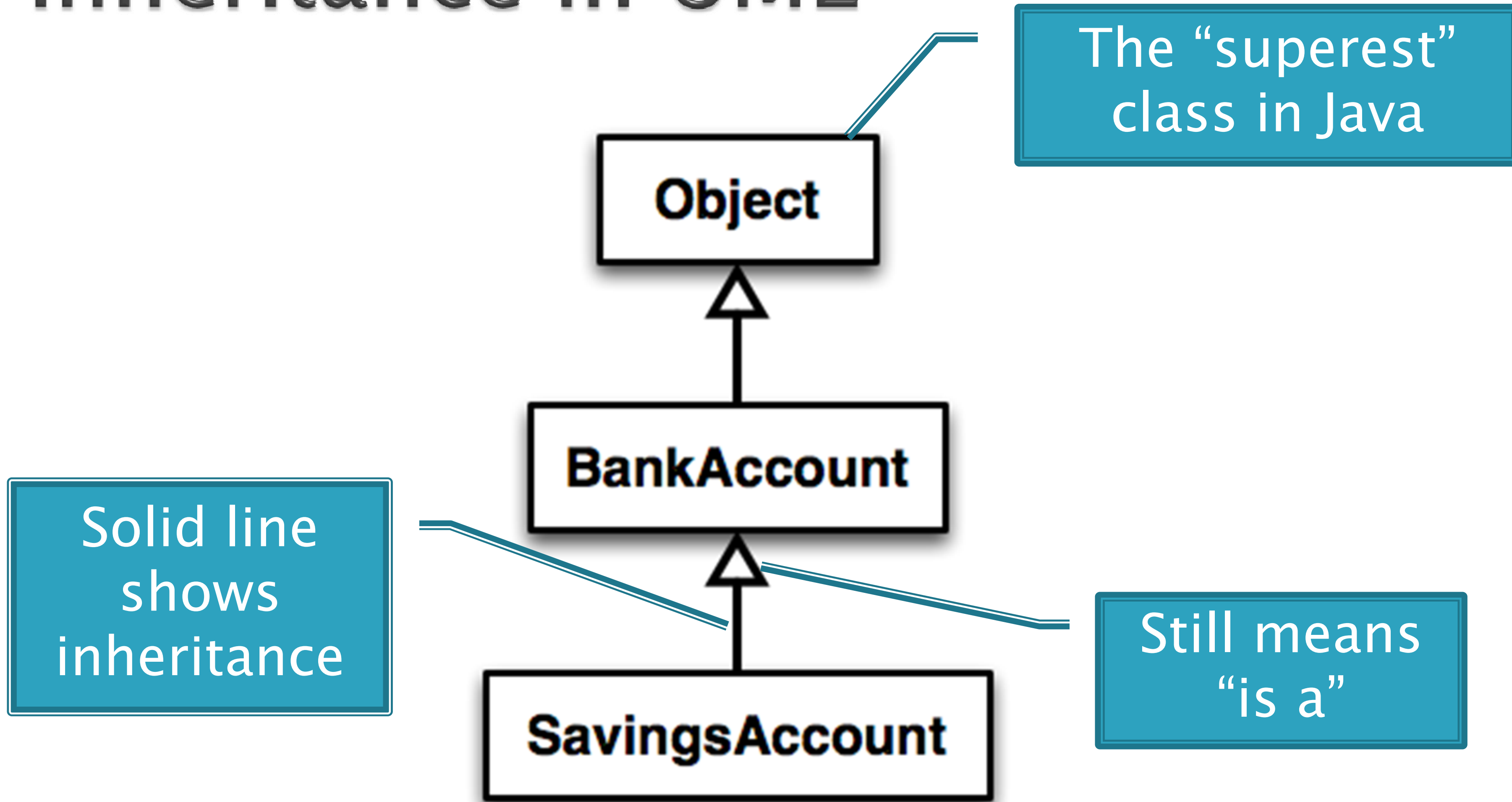
- A Sophomore IS-A Student IS-A **Person.**
- A Continent IS-A LandMass
- An HPCompaqNW8440 IS-A Laptop Computer
- An iPod IS-A MP3Player
- A Square IS-A Rectangle

- It is **not** true that a **Continent** IS-A **Country** or vice-versa.
- Instead, we say that a **Continent** HAS-A **Country.**

Examples From the Java API Classes

String	extends	Object
ArrayList	extends	AbstractCollection
IOException	extends	Exception
BigInteger	extends	Number
BufferedReader	extends	Reader
JButton	extends	Component
MouseListener	extends	EventListener
Frame	extends	Window

Inheritance in UML



Q4

Interfaces vs. Inheritance

- `class ClickHandler implements MouseListener`
  - ClickHandler promises to implement all the methods of MouseListener

For client code reuse

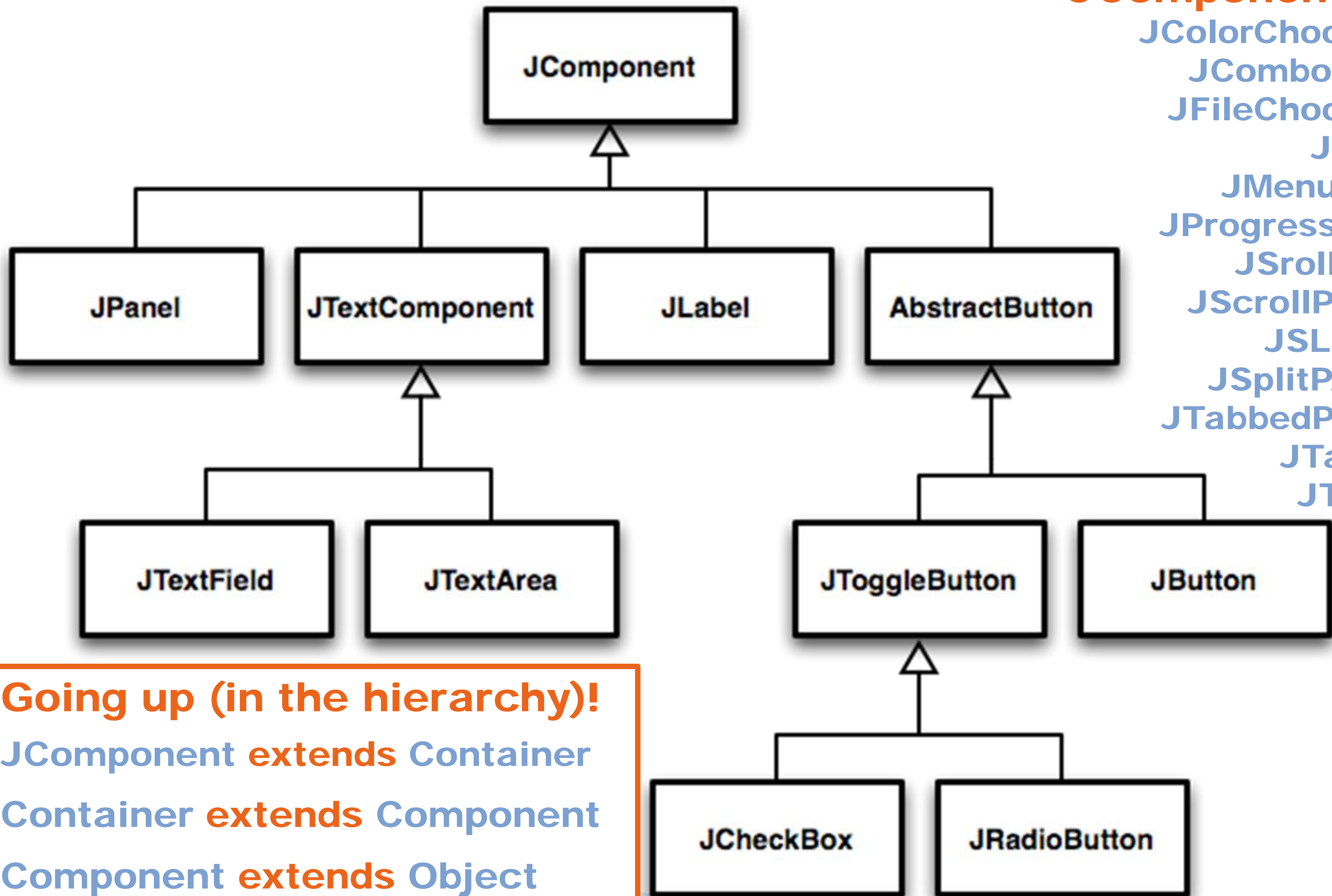
- `class CheckingAccount extends BankAccount`
  - CheckingAccount inherits (or overrides) all the methods of BankAccount

For implementation code reuse

Inheritance Run Amok?

Still more subclasses of JComponent:

JColorChooser
JComboBox
JFileChooser
JList
JMenuBar
JProgressBar
JSrollBar
JScrollPane
JSLider
JSplitPAne
JTabbedPane
JTable
JTree



Going up (in the hierarchy)!

JComponent extends Container

Container extends Component

Component extends Object

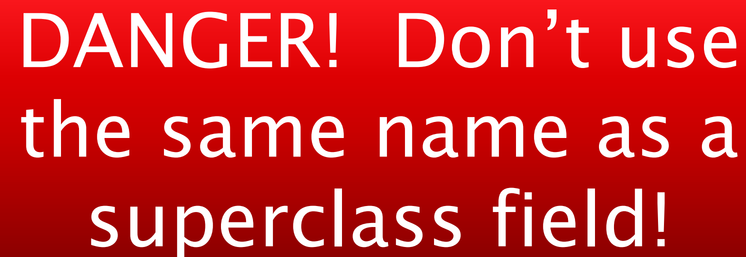
With Methods, Subclasses can:

- Inherit methods unchanged
  - No additional code needed in subclass
- Override methods
  - Declare a new method with same signature to use instead of superclass method
- Partially Override methods
  - call super.sameMethod( ), and also add some other code.
- Add entirely new methods not in superclass

Q5

With Fields, Subclasses:

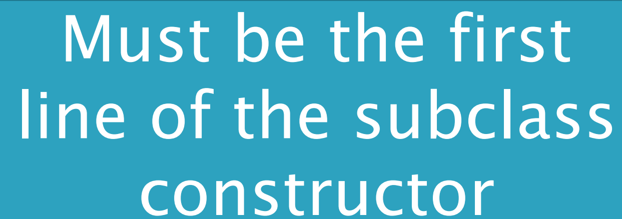
- **ALWAYS inherit** all fields **unchanged**
- **Can add** entirely new fields not in superclass



Q6

Super Calls

- Calling superclass **method**:
  - `super.methodName(args);`
- Calling superclass **constructor**:
  - `super(args);`



Q7

Polymorphism and Subclasses

- A subclass instance is a superclass instance
  - Polymorphism still works!
  - `BankAccount ba = new SavingsAccount();`
    `ba.deposit(100);`

For client code reuse

- But not the other way around!
  - `SavingsAccount sa = new BankAccount();`
    `sa.addInterest();`

BOOM!

- Why not?

Q8

Another Example

- Can use:
  - ```
    public void transfer(double amt, BankAccount o){
        withdraw(amount);
        o.deposit(amount);
    }
    ```
    in BankAccount
- To transfer between different accounts:
  - `SavingsAccount sa = …;`
  - `CheckingAccount ca = …;`
  - `sa.transfer(100, ca);`
    ```

Abstract Classes

- Halfway between superclasses and interfaces
  - Like regular superclass:
    - Provide implementation of some methods
  - Like interfaces
    - Just provide signatures and docs of other methods
    - Can't be instantiated
- Example:
  - ```
    public abstract class BankAccount {
         /** documentation here */
         public abstract void deductFees();
         …
    }
    ```

    ```

# Access Modifiers

- Review
  - **public**—any code can see it
  - **private**—only the class itself can see it

- Others
  - **default** (i.e., no modifier)—only code in the same **package** can see it
    - good choice for classes
  - **protected**—like default, but subclasses also have access
    - sometimes useful for helper methods

Q8

# Inheritance and Abstract Class Example

- Shape Hierarchy

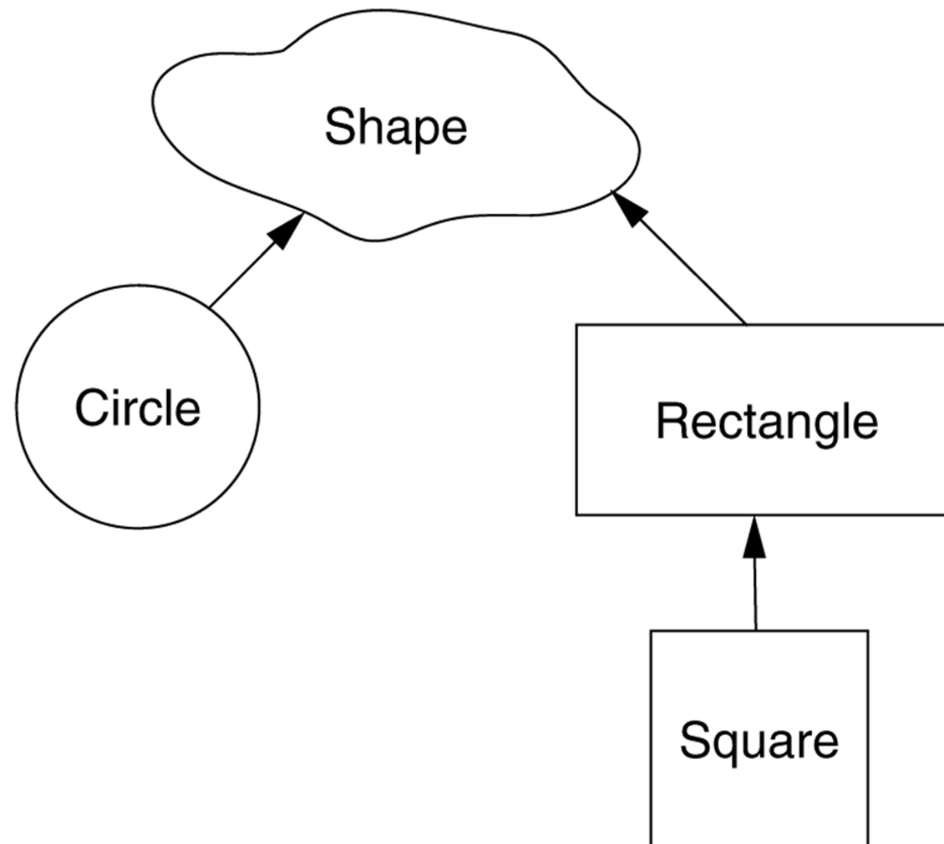
  From Weiss: *Data Structures and Problem Solving Using Java*

  *Code is in Today's project*

# 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.




 © 2002 Addison Wesley

# The Shape Interface

```
public interface Shape extends Comparable {
    public double area();

    public double perimeter();

    public double semiPerimeter();
}
```

AbstractShape class definition

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

    public int compareTo( Object rhs ) {
        double diff = area( ) - ((Shape)rhs).area( );
        if( diff == 0 )
            return 0;
        else if( diff < 0 )
            return -1;
        else
            return 1;
    }

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

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

compareTo is not required to return these specific values. Why does Weiss do it this way?

Circle class definition

```java
public class Circle extends AbstractShape {

    private double radius;

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

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

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

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

Rectangle class definition

```
public class Rectangle extends AbstractShape {

    private double length;
    private double width;

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

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

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

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

    public double getLength( ) {
        return length;
    }

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

implements the abstract methods

overrides a method from the Object class

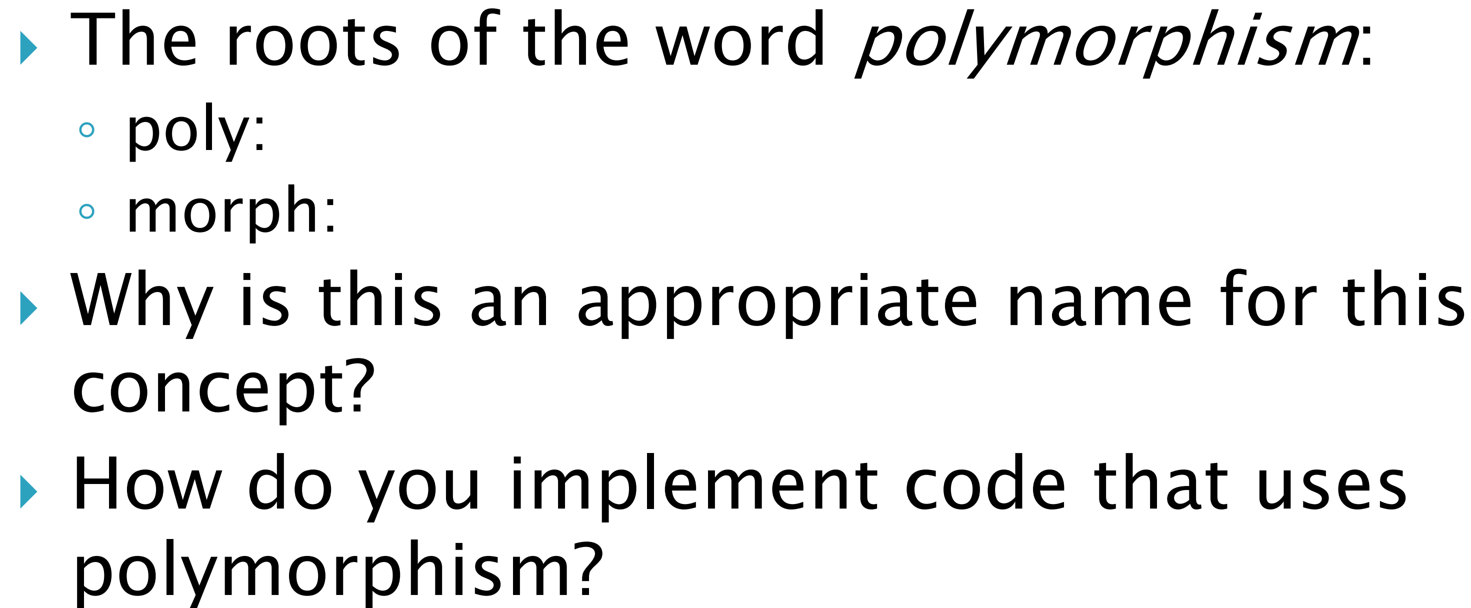
Methods unique to this class

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: " + getLength( );
    }
}
```

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

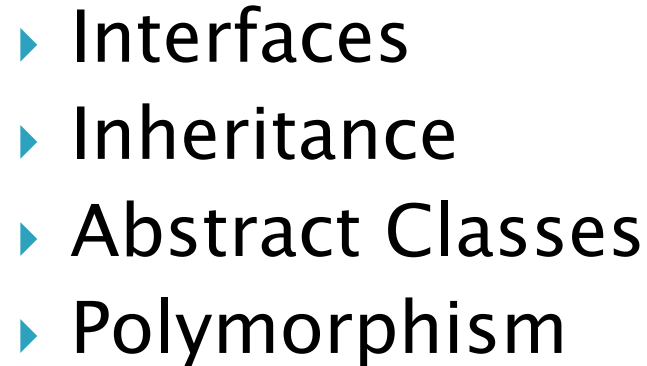
```
public static void printAll( Shape [ ] arr ) {
    for( int i = 0; i < arr.length; i++ )
        System.out.println( arr[ i ] );
}

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
```

More on these topics later



- Interfaces
- Inheritance
- Abstract Classes
- Polymorphism

Hardy's Taxi intro