CENG 1004
Introduction to Object Oriented Programming

Spring 2016

WEEK 6

Today's Topics

- Lecture 5 Review
- Polymorphism
- Interfaces

Lecture 5 Review

Inheritance

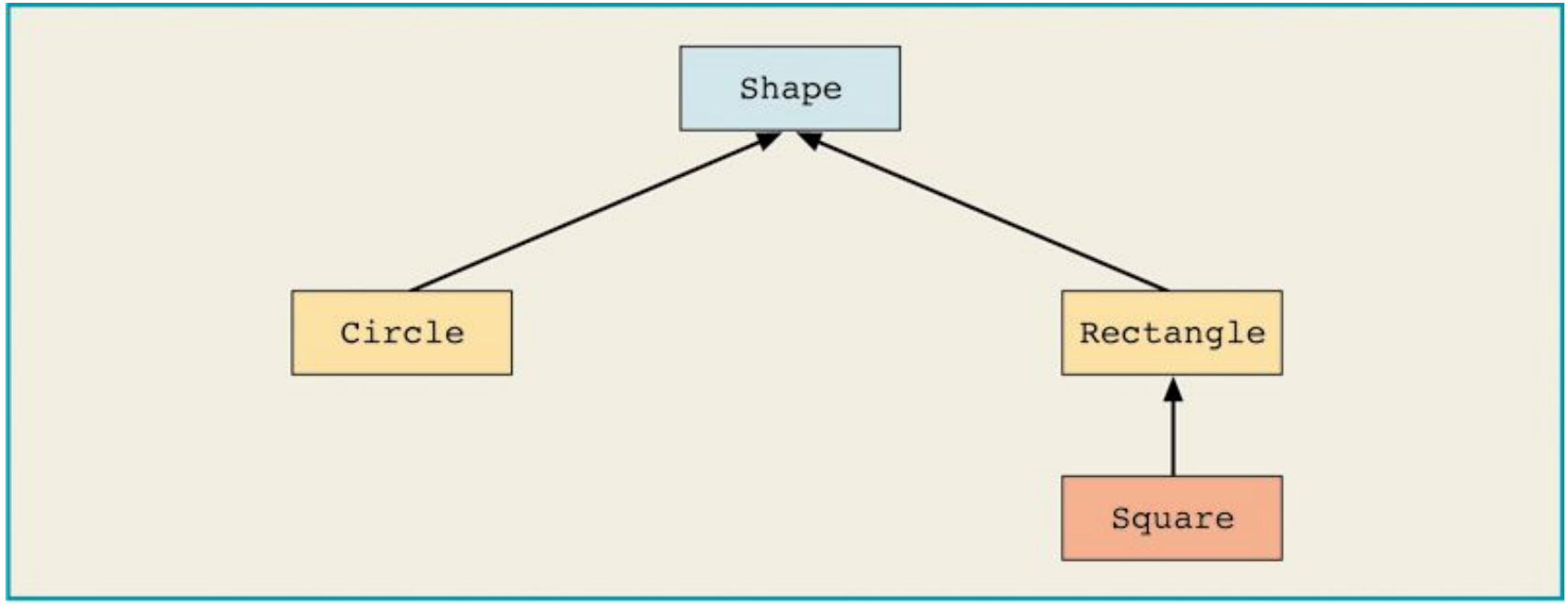


Figure 11-1 Inheritance hierarchy

```
public class ClassName extends ExistingClassName
{
    memberList
}
```

UML Class Diagram: `class` Box

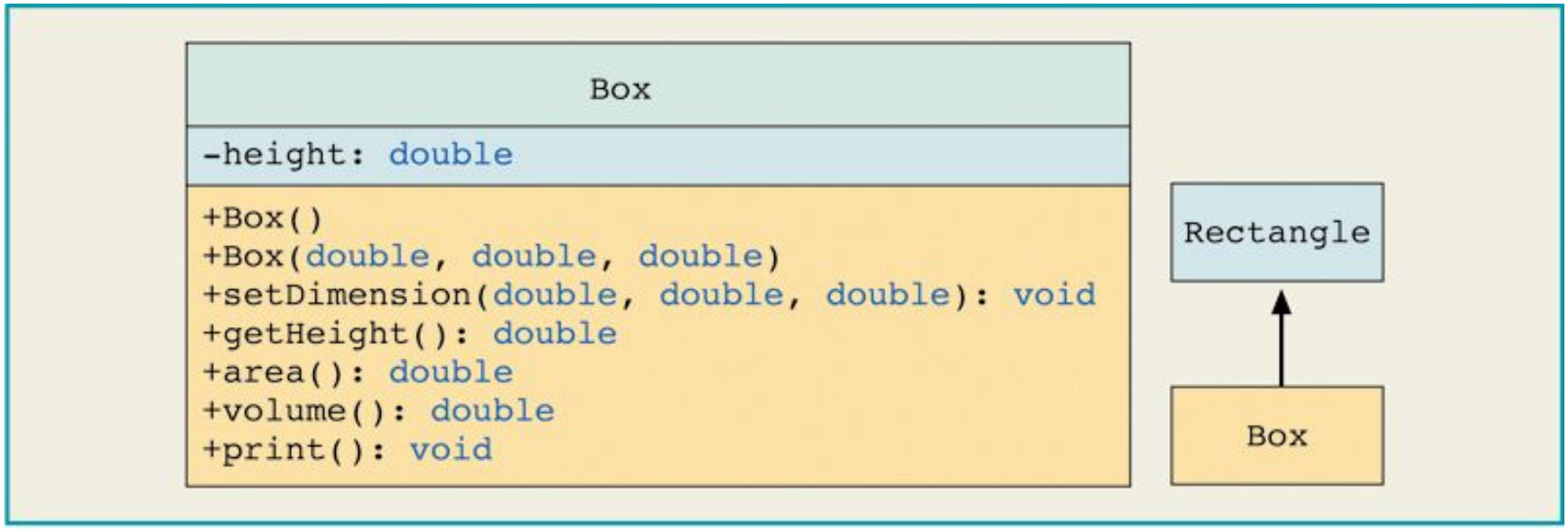


Figure 11-3 UML class diagram of the `class` Box and the inheritance hierarchy

Both a Rectangle and a Box have a surface area, but they are computed differently

Overriding Methods

- A subclass can override (redefine) the methods of the superclass
  - Objects of the subclass type will use the new method
  - Objects of the superclass type will use the original

class Rectangle

```
public double area()
{
    return  getLength() * getWidth();
}
```

class Box

```
public double area()
{
    return  2 * (getLength() * getWidth()
                 + getLength() * height
                 + getWidth() * height);
}
```

final Methods

- Can declare a method of a class final using the keyword `final`

```
public final void doSomeThing()
{
      //...
}
```

- If a method of a `class` is declared `final`, it cannot be overridden with a new definition in a derived class

Modifiers

- A subclass does not inherit/access the **private** members of its parent class.

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Hiding Fields

- Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different.

- Hiding fields is not recommended as it makes code difficult to read.

Calling methods of the superclass

- To write a method's definition of a subclass, specify a call to the public method of the superclass
  - If subclass overrides public method of superclass, specify call to public method of superclass:

    ```
    super.MethodName(parameter list)
    ```

  - If subclass does not override public method of superclass, specify call to public method of superclass:

    ```
    MethodName(parameter list)
    ```

Defining Constructors of the Subclass

- Call to constructor of superclass:
  - Must be first statement
  - Specified by super parameter list

```
public Box()
{
    super();
    height = 0;
}

public Box(double l, double w, double h)
{
    super(l, w);
    height = h;
}
```


12

Object as a Superclass

- **`Object`** is the root of the class hierarchy
  - Every *class* has **`Object`** as a superclass
- All classes inherit the methods of Object
  - But may override them

TABLE 3.2
Methods of Class `java.lang.Object`

Method	Behavior
`Object clone()`	Makes a copy of an object.
`boolean equals(Object obj)`	Compares this object to its argument.
`int hashCode()`	Returns an integer hash code value for this object.
`String toString()`	Returns a string that textually represents the object.

Operations Determined by Type of Reference Variable

- Variable can refer to object whose type is a *subclass* of the variable's declared type
- Type of the *variable* determines what operations are legal
- Java is *strongly typed*
  - Compiler always verifies that variable's type includes the class of every expression assigned to the variable

```
Object obj= new Box(5,5,);
obj.area();                    //  compile-time error.
```

Casting Objects

- *Casting* obtains a reference of different, but *matching,* type
- Casting *does not change* the object!
  - It creates an anonymous reference to the object

```
Box box= (Box)obj;
```

- *Downcast:*
  - Cast *superclass* type to *subclass* type
  - Checks *at run time* to make sure it's ok
  - If not ok, throws **`ClassCastException`**

`instanceof` operator

- **`instanceof`** can guard against **`ClassCastException`**

```
Object obj = ...;
if (obj instanceof Box) {
  Box box = (Box)obj;
  int area= box.area();
  ...;
} else {
  ...
}
```

Abstract Methods and Classes

```
public abstract class Shape{

   // declare fields

   // declare nonabstract methods

   abstract void calculateArea();
   abstract void calculatePerimeter();
}
```

Abstract Methods and Classes

```
class Circle extends Shape{
    void calculateArea() {
        ...
    }
    void calculatePerimeter() {
        ...
    }
}
```

Polymorhism

The Problem

```
public class Circle {
    private double radius;
    ...
    public double area(){
        return Math.PI * Math.pow(radius, 2);
    }
}
```

The Problem

```
public class Rectangle {

    double width;
    double height;
    ...
    public double area(){
        return height * width;
    }
}
```

The Problem

```
public class Drawing {

    ArrayList<Circle> circles = new ArrayList<Circle>();
    ArrayList<Rectangle> rectangles = new ArrayList<Rectangle>();

    public double calculateTotalArea(){
        double totalArea = 0;

        for (Circle circle : circles){
            totalArea += circle.area();    // totalArea = totalArea + circle.area();
        }

        for (Rectangle rect : rectangles){
            totalArea += rect.area();     // totalArea = totalArea + circle.area();
        }
        return totalArea;
    }
}
```

The Problem

- We are asked to introduce a new shape class to the application.

- How Drawing class will be affected?

New Shape: Square

```
public class Square {
    private double side;

    ...

    public double area(){
        return Math.pow(side, 2);
    }
}
```

Drawing

```
public class Drawing {

    ArrayList<Circle> circles = new ArrayList<Circle>();
    ArrayList<Rectangle> rectangles = new ArrayList<Rectangle>();
    ArrayList<Square> squares = new ArrayList<Square>();
    public double calculateTotalArea(){
        double totalArea = 0;
        for (Circle circle : circles){
            totalArea += circle.area();    // totalArea = totalArea + circle.area();
        }
        for (Rectangle rect : rectangles){
            totalArea += rect.area();     // totalArea = totalArea + circle.area();
        }
        for (Square sq : squares){
            totalArea += sq.area();
        }
        return totalArea;
    }
}
```

Design Principle

- Classes should be open for extension, but closed for modification
- Allow classes to be easily extended to add new behaviour without modifying existing code
- How can we accomplish this?

Drawing (Version 2)

```
public class DrawingV2 {

    ArrayList shapes = new ArrayList();

    public double calculateTotalArea(){
        double totalArea = 0;

        for (Object shape : shapes){
            if (shape instanceof Circle){
                Circle circle = (Circle) shape;
                totalArea += circle.area();
            }else if (shape instanceof Rectangle){
                Rectangle rect= (Rectangle) shape;
                totalArea += rect.area();
            }
        }
        return totalArea;
    }
}
```

Problems with Casting

```
Rectangle r = new Rectangle(5, 10);
Circle c = new Circle(5);

Object s = c;

((Rectangle)s).changeWidth(4);
```

- Does this work?

Problems with Casting

- The following code compiles but an exception is thrown at runtime

```
Rectangle r = new Rectangle(5, 10);
Circle c = new Circle(5);
Object s = c;
((Rectangle)s).changeWidth(4);
```

- Casting must be done carefully and correctly
- If unsure of what type object will be then use the **`instanceof`** operator

instanceof

```
Rectangle r = new Rectangle(5, 10);
Circle c = new Circle(5);
Object s = c;
if(s instanceof Rectangle)
      ((Rect)s).changeWidth(4);
```

- syntax: **expression instanceof ClassName**

Casting

- It is always possible to convert a subclass to a superclass. For this reason, explicit casting can be omitted. For example,
  - `Circle c1 = new Circle(5);`
  - `Object s = c1;`

is equivalent to

  - `Object s = (Object)c1;`

- Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.
  - `Circle c2 = (Circle) s;`

Modification to handle Square

```
public class DrawingV2 {
	ArrayList shapes = new ArrayList();
	public double calculateTotalArea(){
		double totalArea = 0;
		for (Object shape : shapes){
			if (shape instanceof Circle){
				Circle circle = (Circle) shape;
				totalArea += circle.area();
			}else if (shape instanceof Rectangle){
				Rectangle rect= (Rectangle) shape;
				totalArea += rect.area();
			}else if (shape instanceof Square){
				Square sq= (Square) shape;
				totalArea += sq.area();
			}
		}
		return totalArea;
	}
}
```

DrawingV2

- Still requires modification to handle new Shapes
- It is possible to add other Objects to the shape list.
  - drawing.add(new String("abc"));
- The common super class for Rectangle, Circle and Square is java.lang.Object

Shape Class

```
public class Shape {

    public double area(){
        return 0;       //default implementation
    }

    public double perimeter(){
        return 0;       //default implementation
    }

}
```

Circle extends Shape

```
public class Circle extends Shape{
    private double radius;
    ...
    public double area(){
        return Math.PI * Math.pow(radius, 2);
    }
}
```

Rectangle extends Shape

```
public class Rectangle extends Shape{

    double width;
    double height;
    ...
    public double area(){
          return height * width;
    }
}
```

Drawing (Version 3)

```
public class DrawingV3 {

    ArrayList<Shape> shapes = new ArrayList<Shape>();

    public void addShape(Shape shape){
        shapes.add(shape);
    }

    public double calculateTotalArea(){
        double totalArea = 0;
        for (Shape shape : shapes){
            totalArea += shape.area();
        }
        return totalArea;
    }
}
```

DrawingV3

- Does not need modification to handle new Shapes
- Only Shape typed objects can be added. Following is not possible now

  drawing.add(new String(" ")); //compile-time error
- What happens if a developer forgets to override area method in a new Shape class?

```
public class Square extends Shape{
    private double side;

    public Square(double side){
        this.side = side;
    }

}
```

Abstract Shape

```
public abstract class Shape {

   public abstract double area();

   public abstract double perimeter();

}
```

Polymorphism

- The term *polymorphism* literally means "having many forms"
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method invoked through a polymorphic reference can change from one invocation to the next

Polymorphism

- Suppose we create the following reference variable:

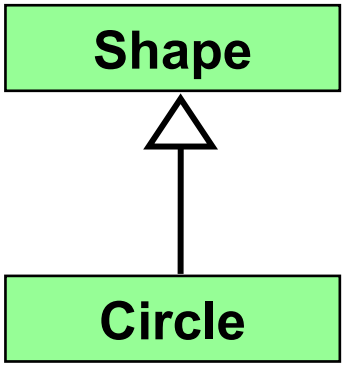
```
Shape shape;
```

- Java allows this reference to point to an `Shape` object, or to any object of any compatible type
- This compatibility can be established using inheritance or using interfaces
- Careful use of polymorphic references can lead to elegant, robust software designs

Polymorhism

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the `Shape` class is used to derive a class called `Circle`, then a `Shape` reference could be used to point to a `Circle` object




```
Shape shape;
shape = new Circle(5);
```

References and Inheritance

- Assigning a child object to a parent reference is called upcasting, and can be performed by simple assignment

  Shape shape;

  shape = new Circle(5);

- Assigning a parent object to a child reference can be done also, but it is called downcasting and must be done manually

  Circle c2 = (Circle) shape;

Polymorphism via Inheritance

- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Suppose the `Shape` class has a method called `area`, and the `Circle and Rectangle` classes override it
- Now consider the following invocation:

```
shape.area();
```

- If `shape` refers to a `Circle` object, it invokes the `Circle` version of `area`; if it refers to a `Rectangle` object, it invokes the `Rectangle` version

New Requirement

- Assume your drawing application also allows user to put text on the drawing area and Drawing class has a new method that draws Shapes and Texts by calling the their draw methods

```
public class Text {

    private String text;

    public Text(String text){
        this.text = text;
    }

    public void draw(){
        //to be imlemented
    }
}
```

New Requirement

- Similarly each Shape class also has draw method

```
public class Circle {
    ...
    public void draw(){
        //to be imlemented
    }
}
public class Rectangle {
    ...
    public void draw(){
        //to be imlemented
    }
}
```

New Requirement

- And we will have a draw metod in Drawing class that can draw all the shapes and texts it contains

```
public class DrawingV4{
    ArrayList<Shape> shapes = new ArrayList<Shape>();
    ArrayList<Text> texts = new ArrayList<Text>();
    ...
    public void draw(){
        //call draw methods of all the shapes and texts
    }
}
```

draw method in Drawing

- And we will have a draw metod in Drawing class that can draw all the shapes and texts it contains

```
public class DrawingV4{
    ...
    public void draw(){
        for (Shape shape : shapes){
            //call draw methods of each shape
        }

        for (Text text : texts){
            text.draw();
        }
    }
}
```

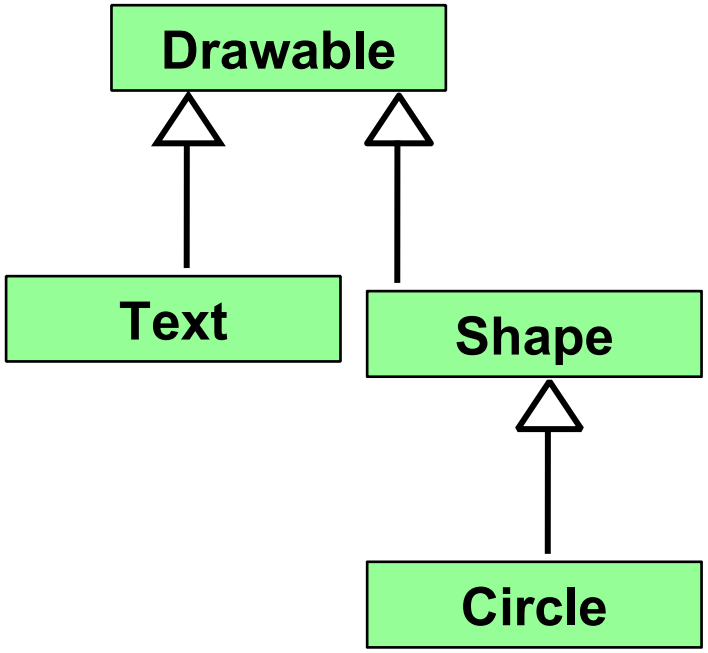
DrawingV4

- We have two lists and in the class to hold drawable objects
  - We have to declare new List for each drawable type
- We have two for loops in the draw method
  - We have to add new loop to draw new drawable types

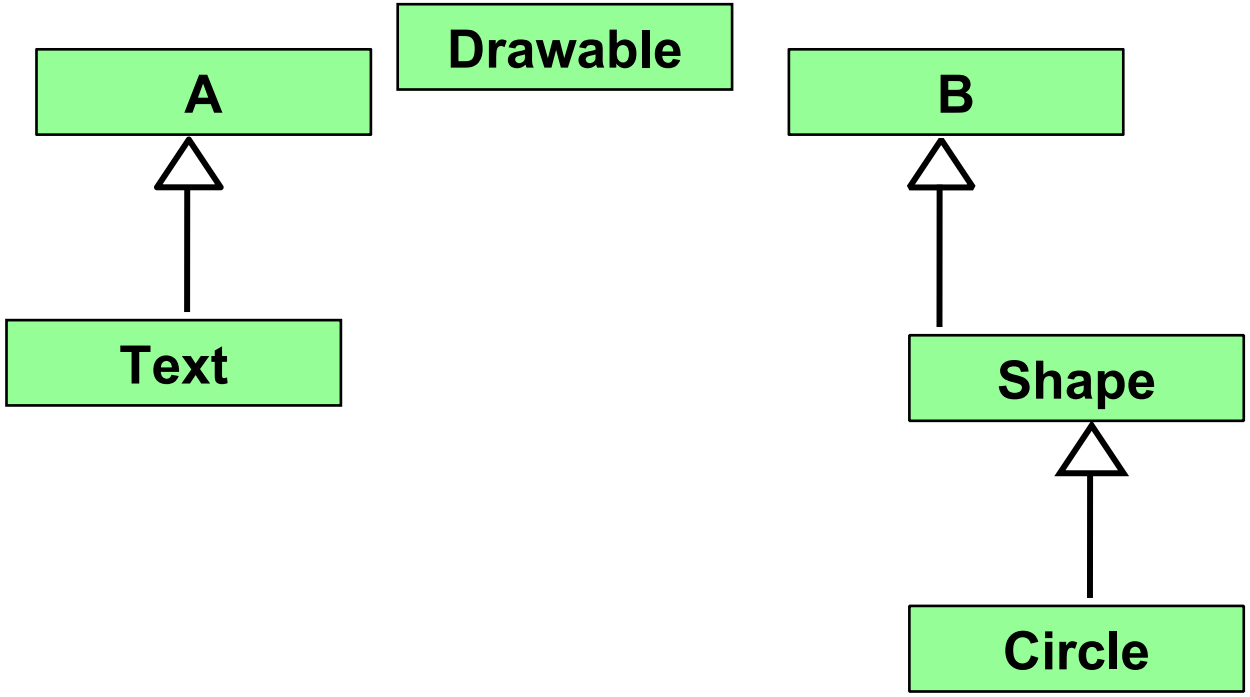
Drawing (Version 5)

```
public class DrawingV4{
    ArrayList<Drawable> drawables= new ArrayList<Drawable>();
        ...
    public void draw(){
        for (Drawable drawable: drawables){
            drawable.draw();
        }
    }
}
```

We can again use inheritance



Assume Text and Shape have already superclasses



- Multiple Inheritance is not allowed in Java

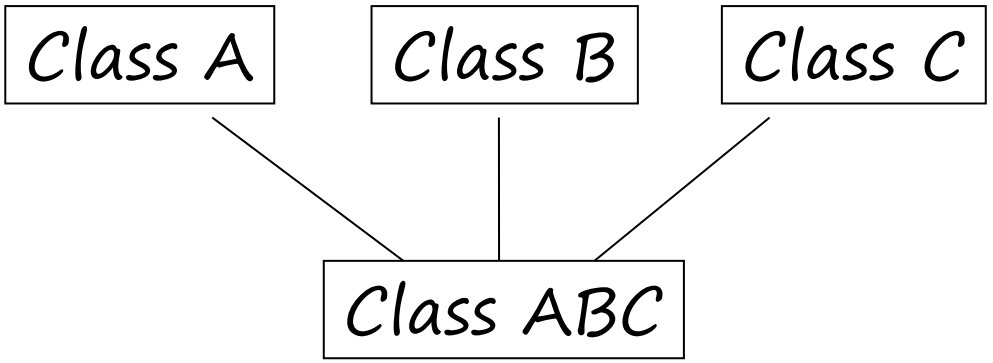
Interface

- An interface is a named collection of method definitions and constants ONLY.
- An interface defines a protocol of behavior that can be implemented by any class anywhere in the class hierarchy.
- An interface defines a set of methods but does not implement them.
- A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behaviors.

Interface and Abstract Classes

- An interface cannot implement any methods, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

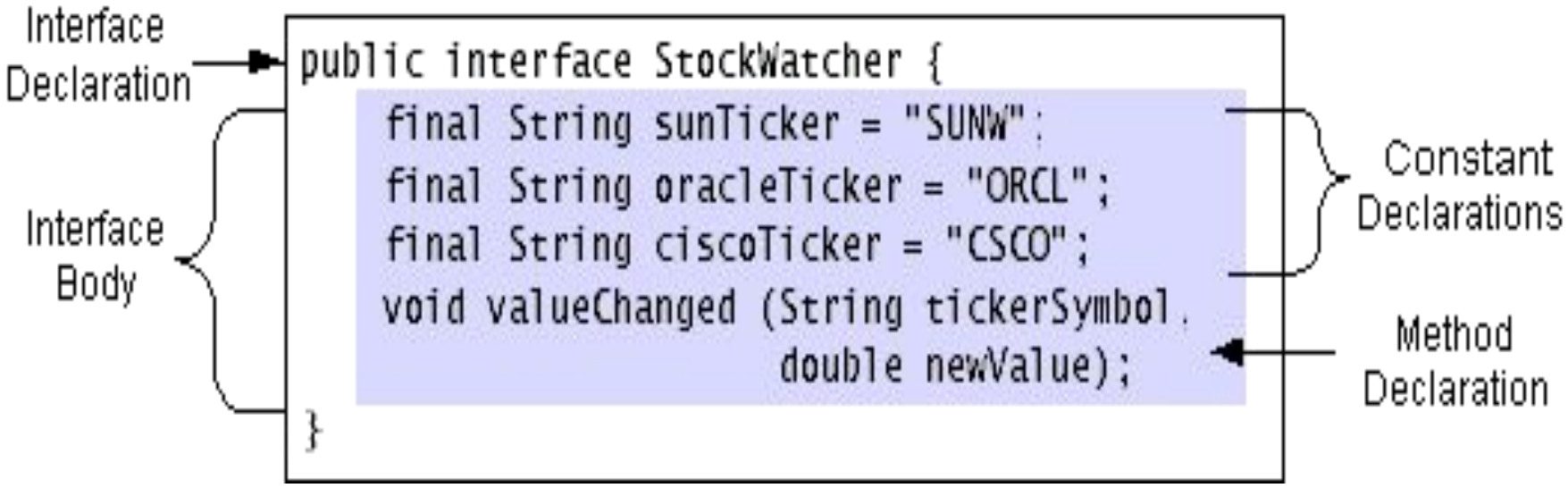
Multiple Inheritance



Class ABC inherits all variables and methods from Class A, Class B, and Class C.

Java does NOT support multiple inheritances.
However, you can use interface to implement the functionality of multiple inheritance.

Defining Interfaces



Interface Declaration

`public`	Makes this interface public.
`interface InterfaceName`	This is the name of the interface.
`Extends SuperInterfaces`	This interface's superinterfaces.

`{`
`    InterfaceBody`
`}`	

public interface StockWatcher{ }

public interface Sortable{ }

Interface Body

- The interface body contains method declarations for **ALL** the methods included in the interface.
- A method declaration within an interface is followed by a semicolon (;) because an interface does not provide implementations for the methods declared within it.
- All methods declared in an interface are implicitly public and abstract.

Implement an Interface

- An interface defines a protocol of behavior.
- A class that implements an interface adheres to the protocol defined by that interface.
- To declare a class that implements an interface, include an implements clause in the class declaration.

Drawable Interface

```
public interface Drawable {

   public void draw();

}
```

Drawable Text

```
public class Text  implements Drawable{

    private String text;

    public Text(String text){
        this.text = text;
    }

    public void draw(){
        //to be imlemented
    }
}
```

Drawable Shape

```
public abstract class Shape implements Drawable{

    public abstract double area();

    public abstract double perimeter();

}
```

Summary for today

- Polymorphism
- Interfaces

References

- http://math.hws.edu/javanotes/
- http://www.cas.mcmaster.ca/~carette/CAS706/2004/presentations/OO_Pujari.ppt
- http://people.cs.pitt.edu/~lorym/CS401/lectures/chapter09.ppt
- http://www.cs.utexas.edu/~cannata/cs345/Class%20Notes/14%20Java%20Upcasting%20Downcasting.htm
- http://www.uwosh.edu/faculty_staff/huen/262/f09/slides/9_Abstract_Interface.ppt