Object Oriented Programming (OOP)

Mohamed Ezz

Lecture 8

Lecture Objectives

- ✓ Understand Data type Casting
- ✓ Differentiate between Upward & downward casting
- ✓ Practice how to override Object Method
- ✓ Define abstract classes and abstract methods.

Review

- ✓ We have two methods overloaded
  - ➢ **f1()**
  - ➢ **f1(int a)**

  which method will be executed when call **f1(5);?**
- ✓ Are the following statement is dynamic or static Binding?
  - ➢ **Shape s= new Rectangle();**
- ✓ State three use for polymorphism

Review: Shape Polymorphism Example

```
class Shape{
    protected int color = 0;
    public void setColor(int color){
            this.color=color;
    }
    public int getColor(){
            return color;
    }
 public float computeArea (){
            return 0;
    }
}
```

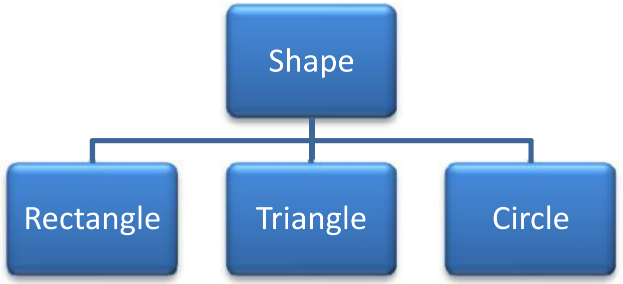
```
class Circle extends Shape{
    private int radius = 0;

  public Circle (int r){
            radius =r;
    }

 public float computeArea (){
            return 22 /7* radius* radius;
    }

 public void doubleSize (){
            radius= 2 * radius;
    }

}
```



Review: Shape Polymorphism Example

```
class Triangle extends Shape{
     private int base = 0;
     private int height = 0;

    public Triangle (int h, int b){
                 base=b;
                 height=h;
     }
 public float computeArea (){
                 return 0.5 * base * height;
     }
}
```

```
class Rectangle extends Shape{
     private int width = 0;
     private int height = 0;

    public Rectangle(int h, int w){
                 width=w;
                 height=h;
     }

    public float computeArea (){
                 return width* height;
     }

 public void swap(){
                 int i= width;
                  width = height;
                  height = I;
     }
}
```

overriding vs. overloading

- polymorphism overriding vs. overloading
  - Can you tell the differences of these concepts?

Casting

- Converting one data type to another either implicitly or explicitly
- **Implicit casting (assigned to a data type of higher size)**
  - int x = 10;   // occupies 4 bytes
  - double y = x;   // occupies 8 bytes
- **Explicit casting (assigned to a data type of lower size)**
  - double x = 10.5;   // 8 bytes
  - int y = x;   // 4 bytes ; raises compilation error

  So correction by explicit casting
  - int y = **(int)** x;

Casting Upwards

- Objects once created always know their type (class)
- You can assign objects of a subclass to a superclass **reference** without using the cast operator (implicit casting)

  e.g.

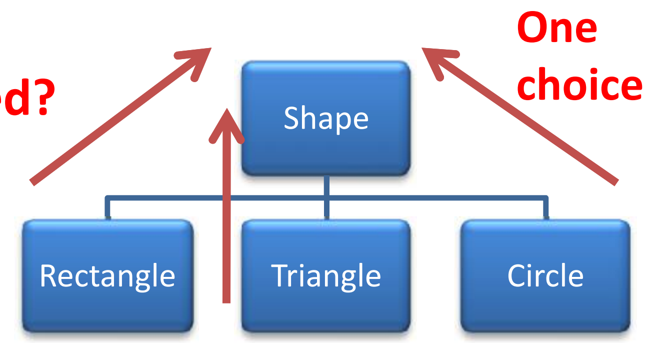
  ```
      Shape s=  new Rectangle(1,2); // no casting required
  ```

  Or

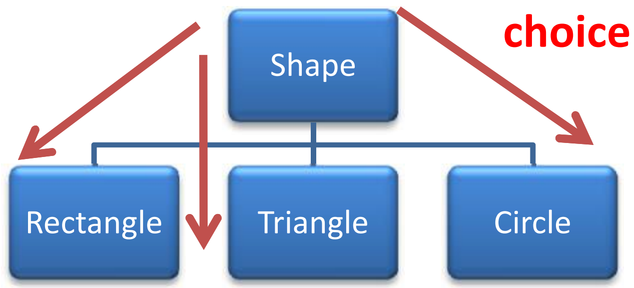
  ```
      Rectangle  r= new Rectangle(1,2) ;
      S=r;  // no casting required
  ```

- The individual objects still know how to perform their behavior

  s.computeArea(); // **which method will be executed?**



Casting Downwards



Multiple choice

- use an explicit casting

  Shape someShape= new Rectangle ()l

  Rectangle rec= someShape; //**syntax error!!**

  Rectangle rec= (Rectangle)someShape; // **Use explicit casting**

- You need to cast downwards to use methods defined only in the derived class type

  someShape.swap(); // **syntax error since swap() is not a method of Shape**

  rec.swap(); **// OK**

- The object itself is **NOT** changed or converted.
  - Casting is telling the compiler to ignore the "type mismatch."
  - Run-time error could occur (**semantic error**).

Casting Run-time error

```
Shape someShape= new Square(2,5);
Rectangle rec= (Rectangle) someShape; //Run-time or Compile-time ERROR?
---------------------------------------------
Shape someShape= new Shape ();
Rectangle rec= (Rectangle) someShape; //Run-time or Compile-time ERROR?
---------------------------------------------
Shape someShape= new Circle (5);
((Rectangle) someShape).swap(); // Run-Time error or Compile-time ERROR?

Shape someShape= new Circle (5);
someShape.doublesize(); // Run-Time error or Compile-time ERROR?

/* The compiler trust you that it will be a Rectangle.  But it create a
   run-time ERROR!! */
```

Avoid Run-time error due to Casting

- Casting downwards to the wrong object is illegal so we use **instanceof** to check the class

```
if (someShape instanceof Rectangle){
    Rectangle rec = (Rectangle) someShape;
    //Or
    ((Rectangle) someShape).swap();
    }
```

Circle reference refer to circle objects

```
Circle c = new Circle(3);
```

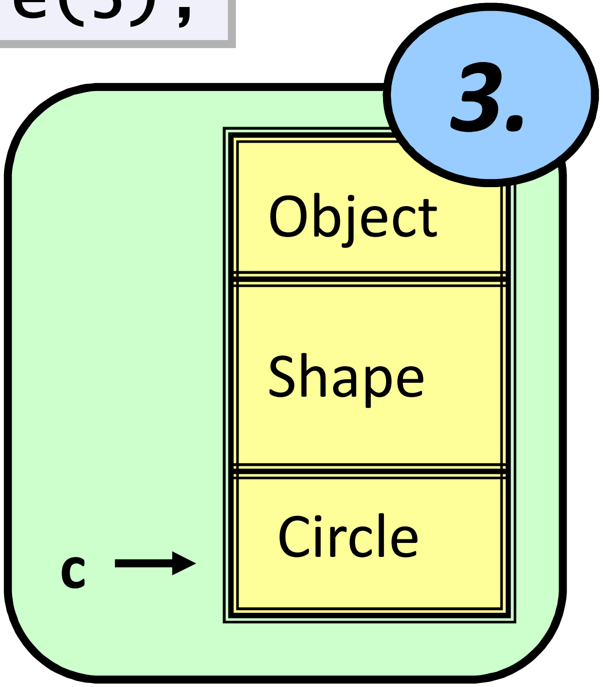
Which statement is correct ?

c.toString();

c.setColor(5);

c.doublesize();

How to correct error?



Shape reference refer to circle objects

```
Shape c = new Circle(3);
```

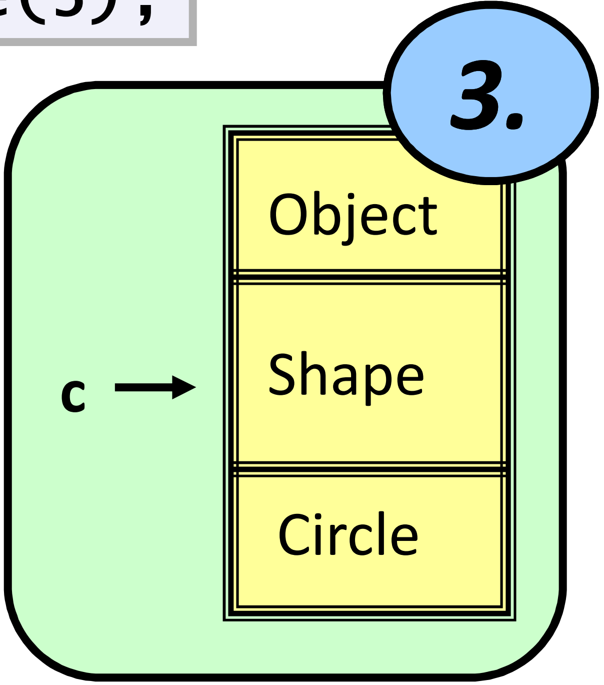
Which statement is correct ?

c.toString();

c.setColor(5);

c.doublesize();

How to correct error?



Object reference refer to circle objects

```
Object c = new Circle(3);
```

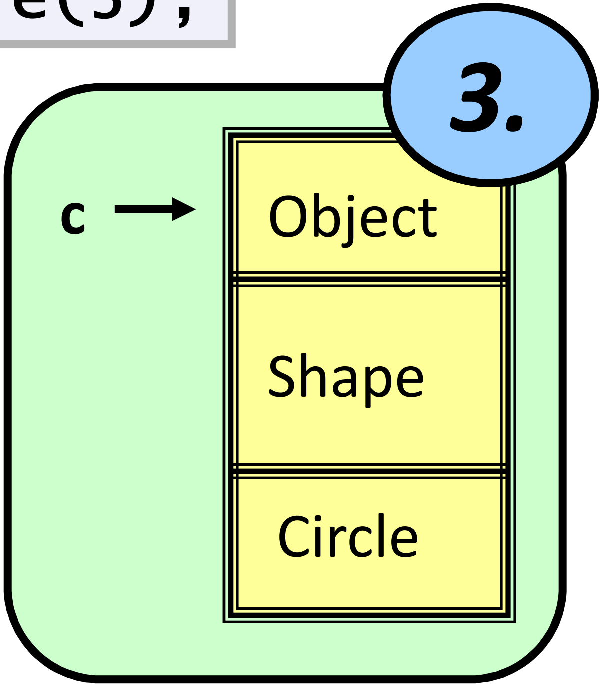
Which statement is correct ?

c.toString();

c.setColor(5);

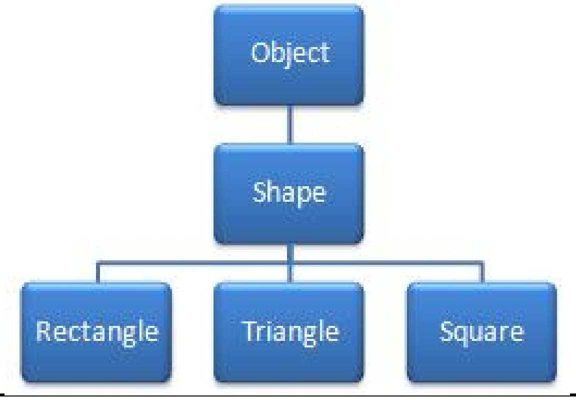
c.doublesize();

How to correct error?



Override Object Class methods

Object class has methods & attribute inherited for all java classes such as:



Methods	Description	Issues
equals(Object o)	compares two objects of same type for equality and returns true if equals and false otherwise	shallow compare which mean primitive attributes compared while object not compared(ref only)
toString()	returns a String representation of an object	doesn't return the member variables in proper format
clone()	takes no arguments and returns a copy of the object on which it is called	shallow copy which mean primitive attributes copied while object not copied (ref only)

Override Object – equals method

```
class Rectangle {
     private int width = 0;
     private int height = 0;
     private Point p= new Point(0,0);

    public Rectangle(int h, int w){
                 width=w;
                 height=h;
     }
public boolean equals (Object o){
                 Rectangle r = (Rectangle )o;
                 if( r.width == width &&
                     r.height == height )
                                 return true;
                 else
                                 return false;
     }
}
```

```
class TestRectangle {

    public static void main(string ar[]){
         Rectangle r1= new Rectangle (1,2);
         Rectangle r2= new Rectangle (1,2);
         Rectangle r3= new Rectangle (1,3);
         System.out.println(r1.equals(r2);
         System.out.println(r1.equals(r3);

    }
}
```

Override Object – clone method

```
class Rectangle {
     private int width = 0;
     private int height = 0;
     private Point p= new Point(0,0);

    public Rectangle(int h, int w){
               width=w;
               height=h;
     }
    public Object clone(){
       Rectangle r= new Rectangle(width,height);
       return r;

   }
}
```

```
class TestRectangle {

    public static void main(string ar[]){
         Rectangle r1= new Rectangle (1,2);

         Rectangle r2= r1.clone();
          r2.p.setX(5);
          r2.p.setY(5);
         System.out.println(r1);
         System.out.println(r2);

    }
}
```

Override Object – toString method

```
class Rectangle {
     private int width = 0;
     private int height = 0;
     private Point p= new Point(0,0);

    public Rectangle(int h, int w){
                 width=w;
                 height=h;
     }
    public String toString(){
        return “width =” + width+ “height =” +
height+ “ x=”+p.x + “ y=”+p.y ;

   }
}
```

```
class TestRectangle {

    public static void main(string ar[]){
         Rectangle r1= new Rectangle (1,2);
         Rectangle r2= r1.clone();
          r2..p.setX(5);
          r2..p.setY(5);
         System.out.println(r1);
         System.out.println(r2);

   }
}
```

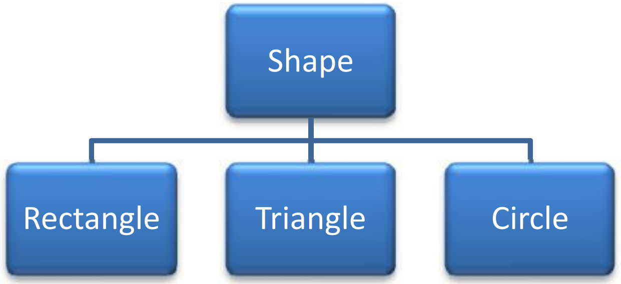
Review: Inheritance

```
class Shape{
    protected int color = 0;
    public void setColor(int color){
              this.color=color;
    }
    public int getColor(){
              return color;
    }
 public float computeArea (){
              return 0;
    }
}
```

Are computeArea method implementation required?

```
class Circle extends Shape{
    private int radius = 0;

  public Circle (int r){
              radius =r;
    }
 public float computeArea (){
              return 22 /7* radius* radius;
    }
 public void doubleSize (){
              radius= 2 * radius;
    }
}
```



What we should do?

- We need to improve the situation by preventing a **developer** from instantiating the **Super** class, because a developer has **marked it as having missing functionality**.
- It also provides **compile-time safety** so that you can ensure that any class that extend your **Super** class provide the bare minimum functionality to work
- Inheritors somehow have to magically know that they **have** to override a method in order to make it work.

Solution: Abstract Class

- This is a class with at least one method without implementation (abstract)
- You can not create instance from that class
- The inherited class from this abstract may implement the abstract methods

Shape Abstract Example

```
abstract class Shape{
    protected int color = 0;
    public void setColor(int color){
            this.color=color;
    }
    public int getColor(){
            return color;
    }
 abstract public float computeArea() ;
// need to be implemented by
//descendent class (child)
}
```

```
class Circle extends Shape{
    private int radius = 0;

   public Circle (int r){
             radius =r;
    }

  public float computeArea (){
            return 22 /7* radius* radius;
    }

  public void doubleSize (){
            radius= 2 * radius;
    }

}
```

Shape Abstract Example cont.

```
class Triangle extends Shape{
    private int base = 0;
    private int height = 0;

   public Triangle (int h, int b){
              base=b;
              height=h;
    }
 public float computeArea (){
              return 0.5 * base * height;
    }
}
```

```
class Rectangle extends Shape{
    private int width = 0;
    private int height = 0;

   public Rectangle(int h, int w){
              width=w;
              height=h;
    }

   public float computeArea (){
              return width* height;
    }

 public void swap(){
              int i= width;
               width = height;
               height = I;
    }
}
```

Abstract Classes in Java

- Abstract classes created using the **abstract** keyword:

  public abstract class Shape{ … }

- In an abstract class, several abstract methods are declared.
  - An abstract method is not implemented in the class, only declared. The body of the method is then implemented in subclass.
  - An abstract method is decorated with an extra "**abstract**" keyword.
- Abstract classes can not be instantiated! So the following is illegal:

  Shape s = new Shape();

Abstract methods Example

```
abstract class Shape{
    protected int color = 0;
    protected Point origin;
    public void setColor(int color){
              this.color=color;
    }
    public int getColor(){
              return color;
    }

   abstract public float computeArea() ;
   abstract public float computePerimeter () ;
  abstract void draw();
  abstract void resize();
 public void move(Point newPlace)  ? abstract

}
```

- Abstract methods are declared but does not contain an implementation.
- So Shape can NOT be instantiated
  (ie., can not be used to create an object.)