COMP2511

Object Oriented Programming (OOP) in Java

Prepared by

Dr. Ashesh Mahidadia

OOP in Java

- Object Oriented Programming (OOP)
- Inheritance in OOP
- Introduction to Classes and Objects
- Subclasses and Inheritance
- Abstract Classes
- Single Inheritance versus Multiple Inheritance
- Interfaces
- Method Forwarding (Has-a relationship)
- Method Overriding (Polymorphism)
- Method Overloading
- Constructors

Subclasses and Inheritance: *First Approach*

We want to implement *GraphicalCircle*.

This can be achieved in at least 3 different ways.

First Approach:

- In this approach we are creating the new separate class for *GraphicalCircle* and **re-writing** the code already available in the class *Circle*.
- For example, we re-write the methods *area* and *circumference*.
- Hence, this approach is NOT elegant, in fact its the worst possible solution.
  Note again, its the worst possible solution!

```
// The class of graphical circles

public class GraphicalCircle {
     int  x, y;
     int  r;
     Color  outline, fill;

     public  double  circumference( ) {
          return 2 * 3.14159 * r ;
     }
     public  double  area ( ) {
          return  3.14159 * r * r ;
     }

     public void draw(Graphics  g) {
          g.setColor(outline);
          g.drawOval(x-r, y-r, 2*r, 2*r);
          g.setColor(fill);
          g.fillOval(x-r, y-r, 2*r, 2*r);
     }
}
```

Subclasses and Inheritance: *Second Approach*

- We want to implement *GraphicalCircle* so that it can make use of the code in the class *Circle*.
- This approach uses "**has-a**" relationship.
- That means, a *GraphicalCircle* has a (mathematical) *Circle*.
- It uses methods from the class *Circle* (*area* and *circumference*) to define some of the new methods.
- This technique is also known as ***method forwarding***.

```
public class GraphicalCircle2 {
    // here's the math circle
    Circle c;
    // The new graphics variables go here
    Color outline, fill;

    // Very simple constructor
    public GraphicalCircle2() {
        c = new Circle();
        this.outline = Color.black;
        this.fill = Color.white;
    }

    // Another simple constructor
    public GraphicalCircle2(int x, int y, int r,
                            Color o, Color f) {
        c = new Circle(x, y, r);
        this.outline = o;
        this.fill = f;
    }

    // draw method , using object 'c'
    public void draw(Graphics g) {
        g.setColor(outline);
        g.drawOval(c.x - c.r, c.y - c.r, 2 * c.r, 2 * c.r);
        g.setColor(fill);
        g.fillOval(c.x - c.r, c.y - c.r, 2 * c.r, 2 * c.r);
    }
```

Subclasses and Inheritance:
Third Approach - Extending a Class

- We can say that *GraphicalCircle* **is-a** *Circle*.
- Hence, we can define *GraphicalCircle* as an ***extension***, or *subclass* of Circle.
- The subclass *GraphicalCircle* ***inherits*** all the variables and methods of its superclass *Circle*.

```
import java.awt.Color;
import java.awt.Graphics;

public class GraphicalCircle extends Circle {

    Color  outline, fill;
    public GraphicalCircle(){
            super();
            this.outline = Color.black;
            this.fill = Color.white;
    }
    // Another simple constructor
    public GraphicalCircle(int x, int y,
                           int r, Color o, Color f){
        super(x, y, r);
        this.outline = o; this.fill = f;
    }

    public void draw(Graphics  g) {
        g.setColor(outline);
        g.drawOval(x-r, y-r, 2*r, 2*r);
        g.setColor(fill);
        g.fillOval(x-r, y-r, 2*r, 2*r);
    }

}
```

Subclasses and Inheritance: Example

We can assign an instance of *GraphicCircle* to a *Circle* variable. For example,

```
GraphicCircle  gc = new GraphicCircle();
      ...
double  area = gc.area();
      ...
Circle  c = gc;
// we cannot call draw method for  "c".
```

Important:

- Considering the variable "**c**" is of type *Circle*,
- we can only access attributes and methods available in the class *Circle*.
- we cannot call *draw* method for "**c**".

Super classes, Objects, and the Class Hierarchy

- Every class has a superclass.
- If we don't define the superclass, by default, the superclass is the class **Object**.

Object Class :

- Its the only class that does not have a superclass.
- The methods defined by Object can be called by any Java object (instance).
- Often we need to **override** the following methods:
  - `toString()`
    - read the API at https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#toString()
  - `equals()`
    - read the API at https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)
  - `hasCode()`

Abstract Classes

Using abstract classes,

- we can declare classes that define only part of an implementation,
- leaving extended classes to provide specific implementation of some or all the methods.

The benefit of an abstract class

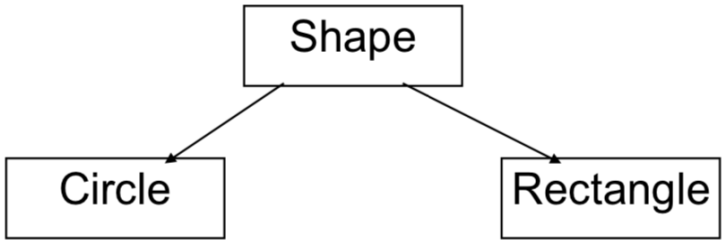
- is that methods may be declared such that the programmer knows the interface definition of an object,
- however, methods can be implemented differently in different subclasses of the abstract class.

Abstract Classes

Some rules about abstract classes:

- An abstract class is a class that is declared abstract.
- If a class includes abstract methods, then the class itself must be declared abstract.
- An abstract class cannot be instantiated.
- A subclass of an abstract class can be instantiated if it overrides each of the abstract methods of its superclass and provides an implementation for all of them.
- If a subclass of an abstract class does not implement all the abstract methods it inherits, that subclass is itself abstract.

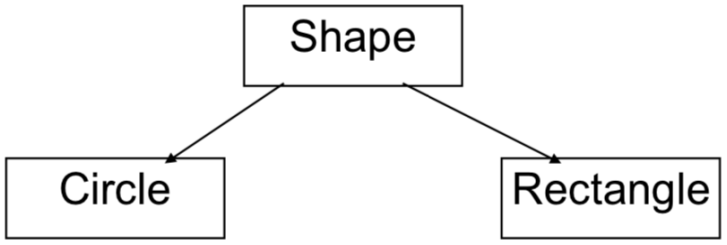
Abstract Class: Example




```
public  abstract class Shape {

    public abstract double area();
    public abstract double circumference();

}
```

```
public class Circle  extends Shape {

    protected static final double pi = 3.14159;
    protected  int  x, y;
    protected  int  r;

   // Very simple constructor
    public Circle(){
        this.x = 1;
        this.y = 1;
        this.r = 1;
    }
    // Another simple constructor
    public Circle(int x, int y, int r){
        this.x = x;
        this.y = y;
        this.r = r;
    }

    /**
     * Below, methods that return the circumference
     * area of the circle
     */
    public  double  circumference( ) {
        return 2 * pi * r ;
    }
    public  double  area ( ) {
        return  pi * r * r ;
    }

}
```

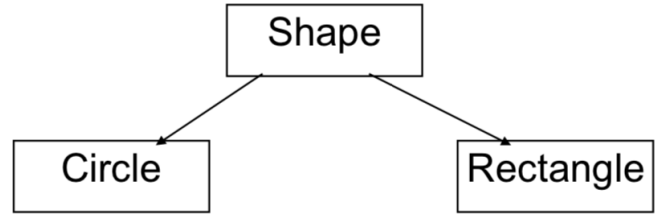
Abstract Class: Example




```
public  abstract class Shape {

    public abstract double area();
    public abstract double circumference();

}
```

```
public class Rectangle extends Shape {

    protected double width, height;

    public Rectangle() {
            width = 1.0;
            height = 1.0;
    }

    public Rectangle(double w, double h) {
         this.width = w;
         this.height = h;
    }

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

    public double circumference() {
            return 2*(width + height);
    }


}
```

Abstract Class: Example



Some points to note:

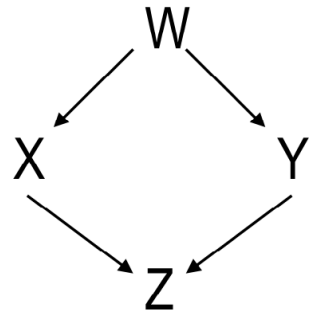
- As Shape is an abstract class, we cannot instantiate it.
- Instantiations of Circle and Rectangle can be assigned to variables of Shape.
  No cast is necessary
- In other words, subclasses of Shape can be assigned to elements of an array of Shape.
  No cast is necessary.
- We can invoke area() and circumference() methods for Shape objects.

We can now write code like this:

```
// create an array to hold shapes
Shape[] shapes = new Shape[4];
shapes[0] = new Circle(4, 6, 2);
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);
shapes[3] = new GraphicalCircle(1, 1, 6,
                Color.green, Color.yellow);

double total_area = 0;
for(int i = 0; i < shapes.length; i++) {
    // compute the area of the shapes
    total_area += shapes[i].area();
}
```

Single Inheritance versus Multiple Inheritance

- In Java, a new class can extend exactly one superclass - a model known as *single inheritance*.
- Some object-oriented languages employ *multiple inheritance*, where a new class can have two or more *super classes*.
- In multiple inheritance, problems arise when a superclass's behaviour is inherited in two/multiple ways.
- Single inheritance precludes some useful and correct designs.
- In Java, interface in the class hierarchy can be used to add multiple inheritance, more discussions on this later.



Diamond inheritance problem