Chapter 10 : Polymorphism

Tasnim Sharmin Alin
Lecturer
Department of Computer Science and Engineeing
Leading University,Sylhet

Principles/Features of OOPs

- **Encapsulation**
- **Inheritance**
- **Polymorphism**

Encapsulation

Encapsulation is:

- Binding the data with the code that manipulates it.
- It keeps the data and the code safe from external interference
- Encapsulation is also known as “**data Hiding**”.

What is Polymorphism

- Polymorphism word comes from ancient Greek where poly means many so polymorphic are something which can take many form.

- ***Polymorphism*** is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations.

- The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Types of Polymorphism

1. **Runtime Polymorhism(or Dynamic polymorphism)**

- ✓ Method overriding is a perfect example of runtime polymorphism.

2. Compile time Polymorhism(or Static polymorphism)

- ✓ Compile time polymorphism is nothing but the method overloading in java.

Abstract class and method

- **Abstraction**

  - **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

  -Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

There are two ways to achieve abstraction in java

- Abstract class
- Interface

Abstract Class

- ❑ A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented.

- ✓ If a class contain any abstract method then class is declared as a abstract class. An abstract class is never instantiated .It is used to provide abstraction.

Syntax :

```
abstract class class_name { }
```

Abstract method

❑ Abstract method

Method that are declared without any body within an abstract class is known as abstract method. The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

Syntax :

```
abstract return_type function_name ();    // No definition
```

Abstract method

❑ Points to remember about abstract method:

- ✓ Abstract method has no body. they just have prototype(method signature).
- ✓ Always end the declaration with a **semicolon**(;).
- ✓ It must be overridden.
- ✓ Abstract method must be in a abstract class.
- ✓ Abstract methods must be implemented in the child class (if the class is not abstract) otherwise program will throw compilation error

Example

```
abstract class Figure{
    double dim1,dim2;
    Figure(double dim1,double dim2){
          this.dim1=dim1;
          this.dim2=dim2;
    }
      abstract double area();
}

class Rectangle extends Figure{
  Rectangle(double dim1,double dim2){
      super(dim1,dim2);
}
    double area(){
        System.out.println("Area of rectangle: ");
        return dim1*dim2;
    }
}
```

Example

```
class Triangle extends Figure{
   Triangle(double dim1,double dim2){
      super(dim1,dim2);
   }
   double area(){
      System.out.println("Area of Triangle: ");
      return (dim1*dim2)/2 ;
   }
}
```

Example

```
public class AbstractAreas {
   public static void main(String[] args) {

      //Figure f = new Figure(10,10); // Illegal now
      Rectangle r = new Rectangle (9,5);
      Triangle t = new Triangle(10,8);

      Figure fig;
      fig=r;
      System.out.println("Area is: "+fig.area());
      fig = t;
      System.out.println("Area is: "+fig.area());
   }
}
```

```
Area of rectangle:
Area is: 45.0
Area of Triangle:
Area is: 40.0
```

Example

```
abstract class Shape {
      public static float pi = 3.142f;
      protected float height;
      protected float width;
      abstract float area();
}
class Square extends Shape {
      Square(float h, float w)  {
             height = h;
             width = w;
      }
      float area()  {
            return height * width;
       }
}
```

Example

```
class Rectangle extends Shape{
     Rectangle(float h, float w)  {
           height = h;
           width = w;
     }
    float area()  {
           return height * width;   }
}
class Circle extends Shape{
     float radius;
     Circle(float r)  {
           radius = r;
     }
     float area() {
          return Shape.pi * radius *radius;
     }
}
```

Example

```
public class AbstractMethodDemo {
    public static void main(String[] args) {
                Square sObj = new Square(5,5);
                Rectangle rObj = new Rectangle(5,7);
                Circle cObj = new Circle(2);
                System.out.println("Area of square : " + sObj.area());
                System.out.println("Area of rectangle : " + rObj.area());
                System.out.println("Area of circle : " + cObj.area());
    }
}
```

Output:

```
Area of square : 25.0
Area of rectangle : 35.0
Area of circle : 12.568
```

When to use Abstract Methods & Abstract Class?

Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods

Interface

The following is a legal interface declaration:

```
        public abstract interface Rollable {
 }
```

Interface

- An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
- An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts.
- A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.

Interface Declaration

- The following interface method declarations are **legal, can compile:**

```
void bounce();
public void bounce();
abstract void bounce();
public abstract void bounce();
abstract public void bounce();
```

- The following interface method declarations **won't compile:**

```
final void bounce();      // final and abstract can never be used together,
and abstract is implied
static void bounce();     // interfaces define instance methods
private void bounce();    // interface methods are always public
protected void bounce();  // (same as above)
```

Java Interface Rules

1. **All interface methods are implicitly public and abstract,** In other words, you do not need to actually type the public or abstract modifiers in the method declaration, but the method is still always public and abstract.
2. **An interface can contain any number of methods**.
3. **All variables defined in an interface must be public, static, and final.**
4. **Interface methods must not be static. Interface methods are abstract, they cannot be marked final, static.**
5. An **interface** can extend one or more other interfaces.
6. An **interface** cannot extend anything except another interface.
7. An **interface** cannot implement another interface or class.
8. An **interface** must be declared with the keyword interface.

Interface

An interface is different from a class in several ways, including:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Importance

- Interfaces are important because they separate what a class does from how it does it.
- Interfaces provide an alternative to multiple inheritance. Java programming language does not support multiple inheritance. But interfaces provide a good solution.
- Any class can implement a particular interface and importantly the interfaces are not a part of class hierarchy. So, the general rule is **extend one but implement many**. A class can extend just one class but it can implement many interfaces. So, here we have multiple types for a class. It can be of the type of its super class and all the interfaces it implements.

example

- Example:

  Let us say we have two interfaces A & B and two classes C & D.

```
interface A{ }
 interface B{ }

class C{ }

class D extends C implements A,B { }
```

So, we can have 3 types for an object of class D as following:

```
A a=new D(); B b=new D(); C c=new D();
```

Example

```
interface box{
  double volume();
}
class boxen implements box{
     double height,depth,width;
     boxen(double h,double w,double d){
              height=h;
              width=w;
              depth=d;
     }
        public double volume(){
              return height*width*depth;
     }
}
```

Example

```
 class boxweight implements box{
       double weight;
      boxweight(double m){
                weight=m;
      }
   public double volume(){
      return weight;  }
}
class shipment implements box{
   double cost;
   shipment(double c){
      cost=c;
   }
   public double volume(){
      return cost;
   }
}
```

Example

```
public class BX2 {
    public static void main(String[] args) {
                boxen b=new boxen(10,20,15);
               System.out.println ("volume is "+b.volume());

               boxweight w=new boxweight(34.6);
               System.out.println ("weight is "+w.volume());

               shipment s=new shipment(50);
               System.out.println ("cost is $: "+s.volume());
    }
}
```

Output:

```
volume is 3000.0
weight is 34.6
cost is $: 50.0
```

Example

```
interface inarea{
     double area();
}

class rectangle implements inarea{
     double height,width;
     rectangle(double h,double w){
               height=h;
               width=w;
     }
     public double area(){
               return height*width;
     }
}

class triangle implements inarea{
     double height,base;
     triangle(double h,double b){
               height=h;
               base=b;
     }
```

Example

```
 public double area(){
             return height*base/2;
     }
}
public class FindAreas {
    public static void main(String[] args) {
              triangle t=new triangle(10,8);
             rectangle r=new rectangle(5,9);

             inarea fig;
             fig=t;
             System.out.println ("area is "+fig.area());
             fig=r;
             System.out.println ("area is "+fig.area());
    }
}
```

output: **area is 40.0**

area is 45.0

Example

```
interface test1{
    final int b=100;
}
interface test2{
    final int b=103;
}
class meth implements test1{
    void show(){
        System.out.println(b+2);
    }
}
public class VarInte implements test1,test2 {
    public static void main(String[] args) {
        meth nn = new meth();
             System.out.println(test1.b);                              output: 100
             System.out.println(test2.b);                                      103
             nn.show();                                                        102
    }
}
```

difference between Abstract Class and Interface

abstract Classes	Interfaces
abstract class can extend only one class or one abstract class at a time	interface can extend any number of interfaces at a time
abstract class can extend from a class or from an abstract class	interface can extend only from an interface
abstract class can have both abstract and concrete methods	interface can have only abstract methods
A class can extend only one abstract class	A class can implement any number of interfaces
In abstract class keyword ‘abstract’ is mandatory to declare a method as an abstract	In an interface keyword ‘abstract’ is optional to declare a method as an abstract
abstract class can have protected , public and public abstract methods	Interface can have only public abstract methods i.e. by default
abstract class can have static, final or static final variable with any access specifier.	interface can have only static final (constant) variable i.e. by default