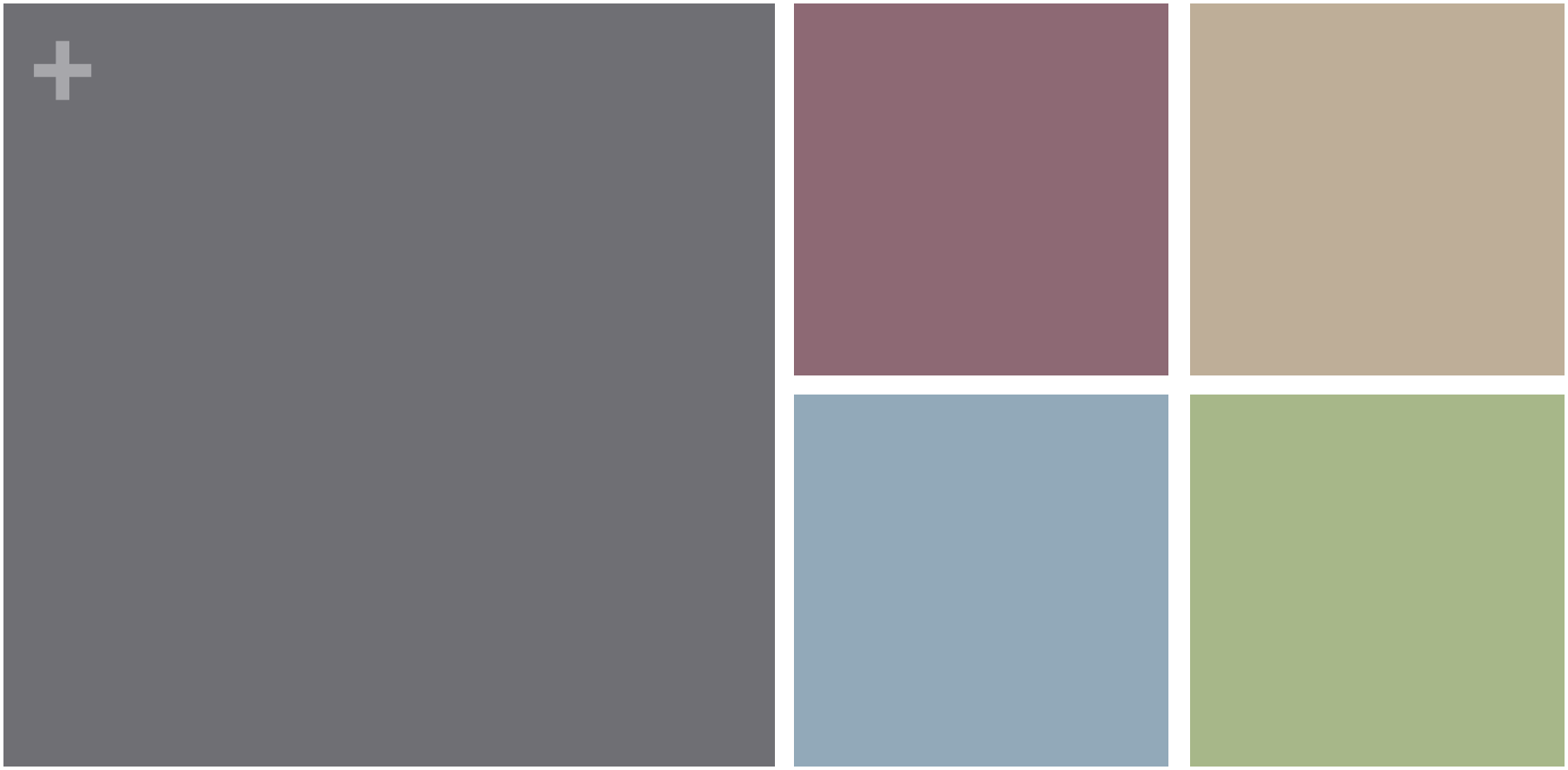


Computer programming (1)

Chapter 6

polymorphism

Agenda

- **polymorphism**
  - Introduction
  - How to achieve Polymorphism in Java ?
  - Types of polymorphism
  - Upcasting
  - Abstract Methods and Classes
  - Interface

polymorphism

- The word polymorphism comes from the Greek for “having many forms.”
- In computer science, polymorphism is when different types of objects respond differently to the same method call.
- Polymorphism allows us to program in the general rather than in the specific.
- Polymorphism is the ability by which, we **can create functions or reference variables which behaves differently in different programmatic context**.
- In other words, the ability to associate many meanings to one method name
  - It does this through a special mechanism known as *late binding* or *dynamic binding*

Late Binding

- The process of associating a method definition with a method invocation is called *binding*
- If the method definition is associated with its invocation when the code is compiled, that is called *early binding*
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called *late binding* or *dynamic binding*

Real life example of polymorphism

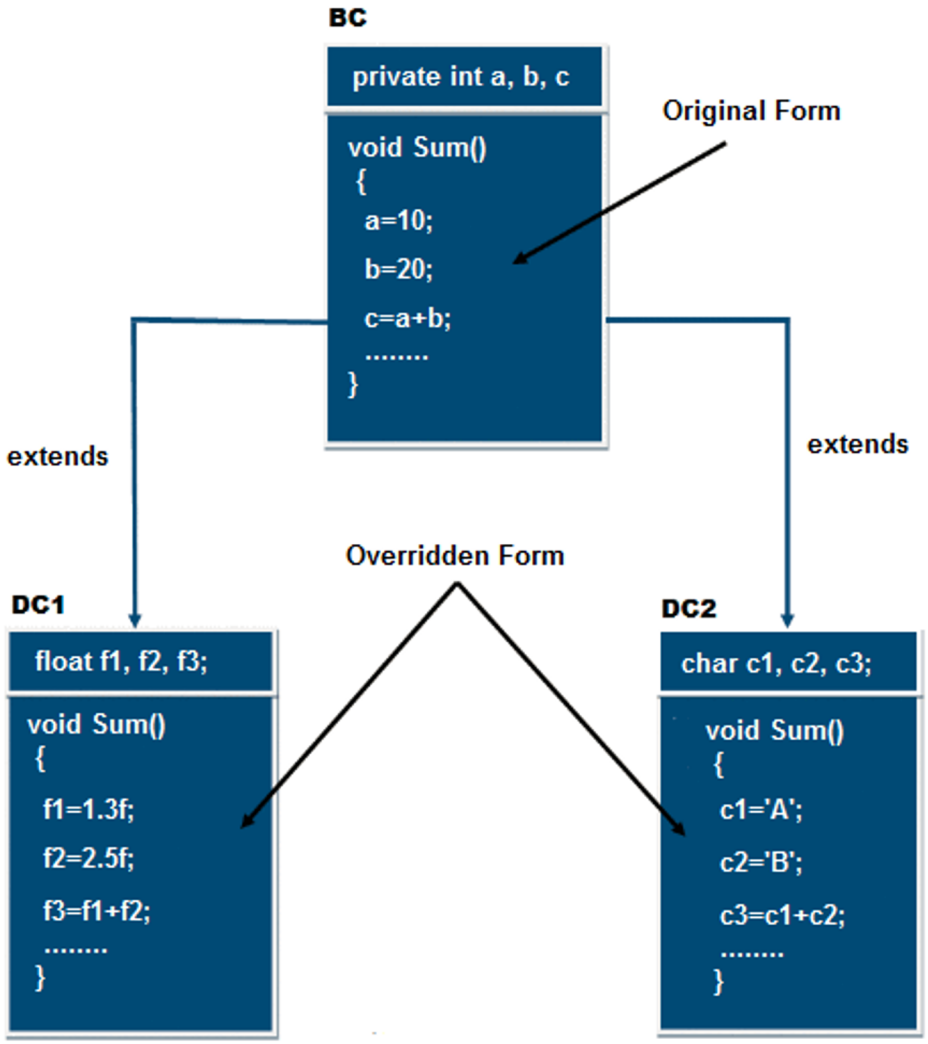
In Shopping malls behave like Customer

In Bus behave like Passenger

In School behave like Student

At Home behave like Son

How to achieve Polymorphism in Java ?



How to achieve Polymorphism in Java ?

- In the above diagram the sum method which is present in BC class is called original form

- The sum() method which are present in DC1 and DC2 are called overridden form,

- Hence Sum() method is originally available in only one form and it is further implemented in multiple forms. Hence Sum() method is one of the polymorphism method.

Types of polymorphism in java

- There are two types of polymorphism in java:
  1. Compile time polymorphism (static binding or method overloading).
  2. Runtime polymorphism (dynamic binding or method overriding)

1- Compile time polymorphism (static binding or method overloading)

- In method overloading, an object can have two or more methods with same name, BUT, with their method parameters different. These parameters may be different on two bases:

1) Parameter type: Type of method parameters can be different. e.g. java.util.Math.max() function comes with following versions:

```
public static double Math.max(double a, double b){..}
public static float Math.max(float a, float b){..}
public static int Math.max(int a, int b){..}
 public static long Math.max(long a, long b){..}
```

- The actual method to be called is decided on compile time based on parameters passed to function in program.

1- Compile time polymorphism (static binding or method overloading)

2) Parameter count: Functions accepting different number of parameters. e.g. in employee management application, a factory can have these methods.

```
EmployeeFactory.create(String firstName, String lastName){...}
EmployeeFactory.create(int id, String firstName, String lastName){...}
```

- Both methods have same name “create” but actual method invoked will be based on parameters passed in program.

overloading example:

```
class Calculation {
  void sum(int a,int b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]) {
    Calculation obj=new Calculation();
    obj.sum(10,10,10);  // 30
    obj.sum(20,20);     //40
  }
}
```

As you can see in the above example that the class has two variance of sum() or we can say sum() is polymorphic in nature since it is having two different forms. In such scenario, compiler is able to figure out the method call at compile-time that's the reason it is known as compile time polymorphism.

Example:

```
class X  {

      void methodA(int num) {
            System.out.println ("methodA: " + num);
      }

      void methodA(int num1, int num2)  {
            System.out.println ("methodA: " + num1 + "," + num2);
     }

     double methodA(double num) {
          System.out.println("methodA: " + num);
          return num;
     }
}
```

```
class Y {
  public static void main (String args []) {
        X Obj = new X();
        double result;
        Obj.methodA(20);
        Obj.methodA(20, 30);
        result = Obj.methodA(5.5);
        System.out.println("Answer is: " + result);
  }
}
```

Output:

methodA: 20

methodA: 20,30

methodA: 5.5

Answer is: 5.5

2- Runtime polymorphism (dynamic binding or method overriding)

- **Runtime polymorphism is essentially referred as method overriding.** Method overriding is a feature which you get when you implement inheritance in your program.

Example 1 of runtime polymorphism

```
class Person {
        void walk() {
                System.out.println("Can Run....");
        }
}

class Employee extends Person {
        void walk() {
                System.out.println("Running Fast...");
        }
        public static void main(String args[]) {
                Person p=new Employee(); //upcasting
                p.walk();
         }
}
```

Output:

Running Fast…

Example2

```
/* File name : Employee.java */
public class Employee {
   private String name;
   private String address;
   private int number;

   public Employee(String name, String address, int number){
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;     }
   public void mailCheck() {
      System.out.println("Mailing a check to " + this.name +
                                 " " + this.address);
   }
 public String getName() {
      return name;
   }}
```

Example2 Con.

```
/* File name : Salary.java */

public class Salary extends Employee {
   private double salary; // Annual salary

public Salary(String name, String address, int number, double salary)
 {     super(name, address, number);
       setSalary(salary);
   }

   public void mailCheck() {
      System.out.println("Within mailCheck of Salary class ");
      System.out.println("Mailing check to " + getName()  + "
                        with salary " + salary);
   }
}
```

Example2 Con.

```
/* File name : VirtualDemo.java */
public class VirtualDemo {

   public static void main(String [] args) {

Salary s   = new Salary("Mohd Moht",  "Ambehta, UP", 3, 3600.00);
Employee e = new Salary("John Adams", "Boston,  MA", 2, 2400.00);

System.out.println("Call mailCheck using Salary reference --");

s.mailCheck();

System.out.println("\n Call mailCheck using Employee reference--")

e.mailCheck();
   }
}
```

output

Constructing an Employee

Constructing an Employee

Call mailCheck using Salary reference --

Within mailCheck of Salary class

Mailing check to Mohd Moht with salary 3600.0

Call mailCheck using Employee reference--

Within mailCheck of Salary class

Mailing check to John Adams with salary 2400.0

Example 3:

```
class Bank{

float getRateOfInterest(){return 0;}

}

class SBI extends Bank{

float getRateOfInterest(){return 8.4f;}

}

class ICICI extends Bank{

float getRateOfInterest(){return 7.3f;}

}

class AXIS extends Bank{

float getRateOfInterest(){return 9.7f;}

}
```

Example 3:

```
class TestPolymorphism{

public static void main(String args[]){

  Bank b;

  b = new SBI();

System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());

  b = new ICICI();

System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());


  b = new AXIS();

System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());


}  }
```

Output:

SBI Rate of Interest: 8.4

ICICI Rate of Interest: 7.3

AXIS Rate of Interest: 9.7

Example 4

```
class Shape{
void draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle...");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle...");}
}
class Triangle extends Shape{
void draw(){System.out.println("drawing triangle...");}
}
```

Example 4

```
class TestPolymorphism2{

public static void main(String args[]){

Shape s;

s = new Rectangle();

s.draw();

s = new Circle();

s.draw();

s = new Triangle();

s.draw();

}

}
```

Output:

drawing rectangle...

drawing circle...

drawing triangle...

Upcasting

- When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:


```
class A{}
class B extends A{}
```

```
A a=new B();//upcasting
```

Method overriding

- **Method overriding** is a perfect example of runtime polymorphism. In this kind of polymorphism, reference of class X can hold object of class X or an object of any sub classes of class X. For e.g. if class Y extends class X then both of the following statements are valid:

```
Y obj = new Y();
//Parent class reference can be assigned to child object
X obj = new Y();
```

overriding example:

```
public class X
{
    public void methodA() //Base class method
    {
        System.out.println ("hello, I'm methodA of class X");
    }
}

public class Y extends X
{
    public void methodA() //Derived Class method
    {
        System.out.println ("hello, I'm methodA of class Y");
    }
}
public class Z
{
   public static void main (String args []) {
        X obj1 = new X(); // Reference and object X
        X obj2 = new Y(); // X reference but Y object
        obj1.methodA();
        obj2.methodA();
   }
}
```

output

```
hello, I'm methodA of class X
hello, I'm methodA of class Y
```

As you can see the methodA has different-2 forms in child and parent class thus we can say methodA here is polymorphic.

Important points:

- Polymorphism is the ability to create a variable, a function, or an object that has more than one form.
- In java, polymorphism is divided into two parts : method overloading and method overriding.

Important points:

- Polymorphism is the ability to create a variable, a function, or an object that has more than one form.
- In java, polymorphism is divided into two parts : method overloading and method overriding.

Abstract Methods and Classes:

Abstract method

- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

Abstract method

- If a class includes abstract methods, the class itself *must* be declared abstract, as in:

```
public abstract class GraphicObject {

   // declare fields

   // declare non-abstract methods

….

  // decalre abstract method

      abstract   void   draw();

    }
```

Abstract Methods and Classes:

Abstract class

- An abstract class is a class that is declared abstract
- Abstract classes cannot be instantiated
- Abstract classes can be subclassed (we can create a subclass from it)
- It may or may not include abstract methods
- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class
- If subclass doesn’t provide implementations then the subclass must also be declared abstract

Can I define an abstract class without adding an abstract method?

- Of course yes. Declaring a class abstract only means that you don't allow it to be instantiated on its own.
- You can't have an abstract method in a non-abstract class.

Abstract Methods and Classes:

- When you subclass an abstract class, the subclass must provide an implementation for each abstract method in the abstract class.
- In other words, it must override each abstract method

Generally

- Abstract classes are useful when you want to create a generic type that is used as the superclass for two or more subclasses,
- But the superclass itself doesn’t represent an actual object. If all employees are either salaried or hourly,
- For example, it makes sense to create an abstract Employee class and then use it as the base class for the SalariedEmployee and HourlyEmployee subclasses.

More about abstract classes

- Not all the methods in an abstract class have to be abstract. A class can provide an implementation for some of its methods but not others. In fact, even if a class doesn't have any abstract methods, you can still declare it as abstract.
- A private method can't be abstract. All abstract methods must be public.

Example of Abstract Classes

```
public abstract class Shape {
    private String name;
    public Shape (String shapeName) {
        name = shapeName;
    }
    public abstract double area( );
    public abstract double perimeter( );
    public String toString( ) {return name; }
}
```

A class with abstract methods must be qualified abstract

Abstract methods – How does one find the area or perimeter of a shape?

There can be no instances of abstract classes. An abstract class provides an interface that concrete classes (such as Circle and Rectangle in this example) can implement.

Example of Abstract Classes

```
public abstract class Shape {
    private String name;
    public Shape (String shapeName) {
        name = shapeName;
    }
    public abstract double area( );
    public abstract double perimeter( );
    public String toString( ) {
        return name;
    }
}
```

Implemented by concrete derived classes that override the abstract methods area( ) and perimeter( )

```
public class Rectangle extends Shape {
    protected double length, width;
    public Rectangle(double len, double w) {
        super("Rectangle");
        length = len;  width = w; }
    public double area( ) {return length * width; }
    public double perimeter( )
      {return 2*(length+width); }
}
```

← Call base constr. first

```
public class Circle extends Shape {
    protected double radius;
    public Circle (double rad) {
        super("Circle");  radius = rad; }
    public double area( )
      {return Math.PI*radius*radius; }
    public double perimeter( )
        {return 2.0* Math.PI * radius; }
}
```

Interface

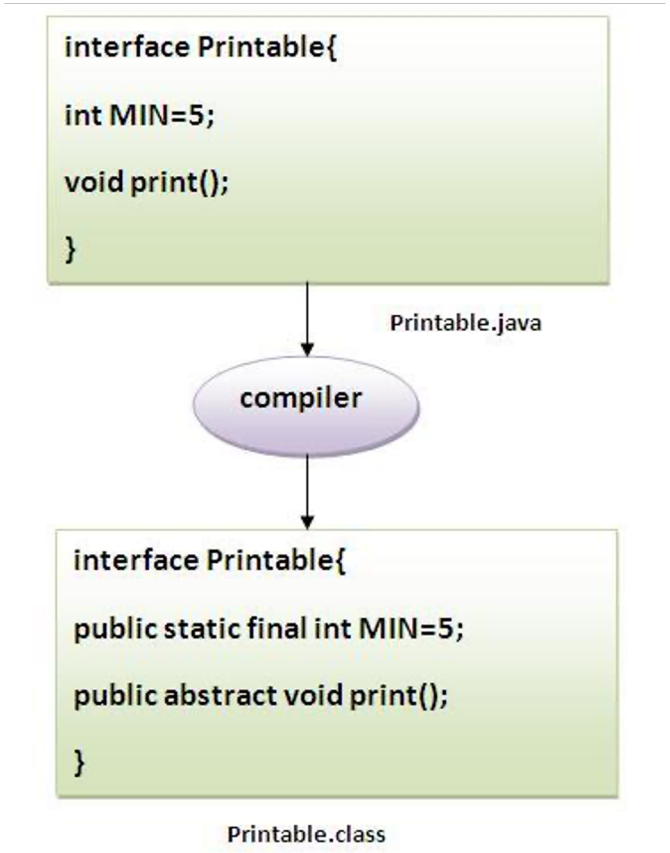
- A Java interface is used to specify what is to be done but not how it is to be done.
  - Java interface consists of only final static variables and abstract methods.
  - No implementation is provided for the methods.

Interface

- An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.
- The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

Interface

- The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.



Simple example of Java interface

```
interface printable{
void print();
}

class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

```
Hello
```

Example1:

```
//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}

}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
      public static void main(String args[]){
      Drawable d=new Circle();//In real scenario, object is provi
                              //ded by method e.g. getDrawable()
      d.draw();
}}
```

Output:

drawing circle

Example2:

```
//TestInterface2.java
interface Bank{
        float rateOfInterest();
}
class SBI implements Bank{
        public float rateOfInterest(){          return 9.15f;       }
}
class PNB implements Bank{
        public float rateOfInterest(){          return 9.7f;        }
}
class TestInterface2{
        public static void main(String[] args){
                Bank b=new SBI();
                System.out.println("ROI: "+b.rateOfInterest());
}}
```

ROI: 9.15

Multiple inheritance in Java by interface

```
interface Printable{
     void print();
}
interface Showable{
     void show();
}
class A7 implements Printable,Showable{
     public void print(){System.out.println("Hello");}
     public void show(){System.out.println("Welcome");}


public static void main(String args[]){
     A7 obj = new A7();
     obj.print();
     obj.show();
  }
}
```

```
Hello
Welcome
```

Difference between abstract class and interface

- Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
3) Abstract class **can have final, non-final, static and non-static variables.**	Interface has **only static and final variables.**
4) Abstract class **can have static methods, main method and constructor.**	Interface **can't have static methods, main method or constructor.**
5) Abstract class **can provide the implementation of interface.**	Interface **can't provide the implementation of abstract class.**
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Abstract Classes versus Interfaces

- Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation.
- If an abstract class contains *only* abstract method declarations, it should be declared as an interface instead.

References

- http://howtodoinjava.com/object-oriented/what-is-polymorphism-in-java/
- http://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/
- http://stackoverflow.com/questions/20783266/what-is-the-difference-between-dynamic-and-static-polymorphism-in-java
- http://howtodoinjava.com/object-oriented/what-is-polymorphism-in-java/
- http://www.javatpoint.com/runtime-polymorphism-in-java

+

Thanks!