Core Java Part - II

1. Object Oriented Programming

- *Encapsulation (Data hiding & Method abstraction)*
- *Inheritance (IS-A Relationship)*
- *Polymorphism*
- *Method Signature*
- *Overloading*
- *Overriding*
- *Constructors*
- *this or super*
- *Final & static*
- *Abstract class and Interface*

OBJECT ORIENTED PROGRAMMING

- ***Encapsulation (Data hiding & Method abstraction)***
- ***Inheritance (IS-A Relationship)***
- ***Polymorphism***
  - ***Method Signature***
  - ***Overloading***
  - ***Overriding***
- ***Constructors***
- ***this or super***
- ***Final & static***
- ***Abstract***
- ***Interface***

Encapsulation

Data hiding

One of the sound object-oriented programming techniques is **hiding the data** within the class by declaring them with **private** accessibility modifier. Making such variables available outside the class through **public setter and getter methods**. Data hiding says "**restrict** access and modification of internal data items through the use of getter and setter methods".

Example:

```
public class Authentication{
        //Data hiding
        private String username;
        private String password;

        public void setUsername(String name){
                if((name != null) && name!="")){
                        username = name;
                }else{
                        Sop("Username cannot be null or empty");
                }
        }
        public String getUsername(){
                return username;
        }
        ….
        ….
}
Authentication obj = new Authentication();
//obj.username="aspire";//Compilation error.
obj.setUsername("aspire");//Ok
obj.setUsername("");//validation error saying "Username cannot be null or empty."
```

Method Abstraction

One of the sound object-oriented programming techniques is “Hiding method implementation complexity from outside users” is called as method abstraction.

```
//Method abstraction
public boolean isValidUser(){
        //Use JDBC API for obtaining connection.
        Connection con = DriverManager.getConnection(“jdbc:oracle:thin:@localhost:1521:XE”, “scott”, “tiger”);
        Check with database whether user is already registered or not.
        If valid user then
                return true;
        else
                return false;
}
```

Encapsulation = Data hiding + Method Abstraction

Encapsulation combines (or bundles) data members with method members together (or the facility that bundles data with the operations that perform on that data is called as encapsulation).

Example:

```
public class Authentication{
        //Data hiding
        private String username;
        private String password;

        public void setUsername(String name){
                if((name != null) && name!=”” ){
                        username = name;
                }
        }
        public void setPassword(String pwd){
                if((pwd != null) && pwd!=”” ){
                        Password = pwd;
                }
        }

        //Method abstraction
        public boolean isValidUser(){
                //Use JDBC API for obtaining connection.
                Connection con = DriverManager.getConnection(“jdbc:oracle:thin:@localhost:1521:XE”, “scott”, “tiger”);
                Check with database whether user is already registered or not.
                If valid user then
                        return true;
                else
                        return false;
```

```
        }
}
```

//Client program

```
Public class AccessSite{
  public static void main(String[] args){
        Authentication auth = new Authentication();
        //set website username and password
        auth.setUsername("aspire");
        auth.setPassword("aspire123");

        boolean success= auth.isValidUser();
        if(success){
                System.out.println("Logged into INBOX…");
        }else{
                System.out.println("Invalid username / password…");
        }
 }
}
```

Inheritance

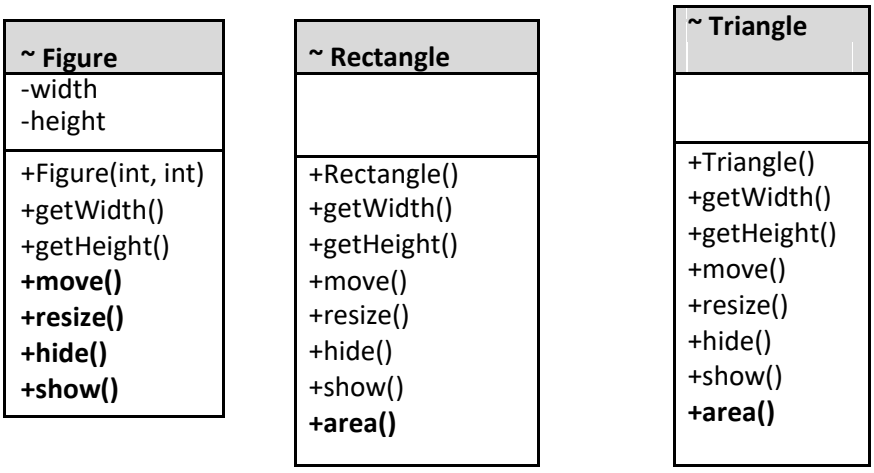
Define common methods in parent class and specific methods in child class.

Example:

```
class Figure{                   //Parent class
        private int width;
        private int height;
        public Figure(int w, int h){
                width = w;
                height = h;
        }
        public int getWidth() {
                return width;
        }
        public int getHeight() {
                return height;
        }
        public void move(int x, int y){
                System.out.println("Move method...");
        }
        public void resize(int width, int height){
                System.out.println("Resize method....");
        }
        public void hide(){
                System.out.println("Hide method...");
        }
        public void show(){
```

```
                System.out.println("show method...");
        }
}
  class Rectangle extends Figure{ //child class
        public Rectangle(int w, int h){
                super(w,h);//Invokes immediate parent class matching constructor
        }
        //subclass specific method.
        double area(){
                return getHeight()* getWidth();
        }
}

class Triangle extends Figure{
        public Triangle(int w, int h){
                super(w,h);
        }
        //subclass specific method.
        double area(){
                Return 0.5 * getHeight()* getWidth();
        }
}
```



Note:

Accessibility modifier	UML symbol
Private	-
Default	~
Protected	#
Public	+

In the above example, Rectangle **IS-A** Figure. Similarly, Triangle **IS-A** Figure. So, inheritance is also called as **IS-A** relationship.

The base class common methods move(), resize(), hide(), and show() are **reused** in the subclasses Rectangle and Triangle. Hence the main advantage with inheritance is Code **REUSABILITY**.

The base class is also called as superclass, supertype, or parent class. The derived class is also called as subclass, subtype, or child class.

In java, the **extends** keyword is used to inherit a superclass.

Example:

```
class Rectangle extends Figure{}
```

In java, a class can inherit at most one superclass. This is called single level inheritance.

Example:

```
class C extends A, B{}          //Compilation error.
```

Conclusion: Java does not support multiple inheritance with classes but supports with interfaces.

The inheritance relationship is represented as **Hierarchical classification**.

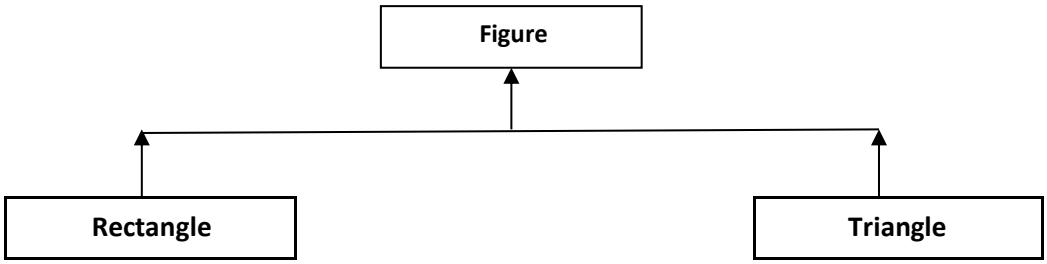


Figure 1: Inheritance hierarchy

The classes higher up in the hierarchy are more **generalized** and the classes lower down in the hierarchy are more **specialized**.

Case 1: Subclass specific methods cannot be invoked using superclass reference variable.

Example:

```
Figure fRef = new Figure(10,20);
fRef.move();
//fRef.area();                  //Compilation error. Undefined method.
```

Case 2: As expected, subtype object is assigned to subtype reference variable:

Example:

```
Rectangle rRef = new Rectangle(10,20);
rRef.move();
rRef.area();            //Compiles and runs without errors.
```

Case 3: The subclass object can be assigned to a superclass reference variable. But, the subclass specific methods cannot be invoked using superclass reference variable, because the compiler only knows reference type methods but not actual runtime object methods.

This assignment involves a widening reference conversion (upcasting).

Example:

```
fRef = new Rectangle(10,30); //Reference type widening. Subtype object is assigned to supertype ref var.
fRef.move Figure ();//Ok
```

fRef.area(); // Compilation error. Undefined method.

Polymorphism

Polymorphism translates from Greek as **many forms** (*poly - many & morph - forms).* Overloading and overriding are the two different types of polymorphisms.

Overloading – Static polymorphism / Compile Time Poly

Overriding – Dynamic polymorphism / Run Time Ploy

Method Signature

It is the combination of the method name and its parameters list. Method signature does not consider return type and method modifiers.

Example:

public int **add(int, int)**{}
public float **add(float, float)**{}

Method overloading

Two or more methods are said to be overloaded if and only if methods with **same name but differs in parameters list** (Differs in Number of parameters, Parameter type, or Parameter order), and without considering return type, method modifiers i.e. Overloaded methods must have different method signature irrespective of their return type, modifiers.

Example:

```
class Addition{
        public int add(int x, int y){}
        public int add(int x, int y, int z){}

        public float add(float x, float y){}
        public double add(double x, double y){}

        public float add(int x, float y){}
        public float add(float x, int y){}
}
```

All above methods are overloaded with each other.

Addition obj = new Addition();
obj.add(10,20);//**compiler** maps with add(int, int)
obj.add(10,20,30); //**compiler** maps with add(int, int, int)

The overloaded methods are resolved by the java compiler at compilation time. Hence overloading is also known as 'Static Polymorphism' or 'Early Binding'.

Method overriding

A subclass can re-implement superclass methods to provide subclass specific implementation. The method in superclass is called as **overridden** method, whereas method in subclass is called as **overriding** method.

Method overriding rules:

a) The accessibility modifier of subclass method must be same or less restrictive. **AND**
b) The superclass and subclass methods must have same return type. **AND**

c) The superclass and subclass method signature must be same (i.e., method name, the number of parameters, parameter types, parameter order). **AND**

d) The throws clause in overriding method can throw all, none or a subset of the checked exceptions which are specified in the throws clause of the overridden method, but not more checked exceptions.

Example:

```
class Figure {
        int width;
        int height;
        Figure(int w, int h) {
                width = w;
                height = h;
        }
        //overridden method
        public double area(){
                return 0.0; //unknow
        }
}
class Rectangle extends Figure {
        Rectangle(int w, int h) {
                super(w, h);
        }
        //overriding method
        public double area() {
                return width * height;
        }
}
public class OverridingDemo {
        public static void main(String[] args) {
                //superclass object assigned to superclass reference variable
                Figure fRef = new Figure();
                Sop(fRef.area()); //0.0            //area() from Figure class

                //subclass object assigned to subclass reference
                variable Figure fRef1 = new Rectangle(10,20);
                Sop(fRef1.area()); //200.0        //area() from Rectangle class.
        }
}
```

The method overriding is resolved at runtime based on **actual object type** but not Reference type.
Hence method overriding is also known as Late Binding or dynamic polymorphism.

Dynamic Method Dispatch (DMD):

Method overriding is also known as Dynamic Method Dispatching.

Example:

```
class Figure{
```

```
        void area(){}
}
class Rectangle extends Figure{
        void area(){}
}
 class Triangle extends Figure{
        void area(){}
}

Figure fRef = new Rectangle(10,20);
fRef.area();                    //area() method from Rectangle object is invoked.


fRef = new Triangle(10,20);
fRef.area();                    //area() method from Triangle object is invoked.
```

The method overriding is also called as Dynamin Method Dispatching, Dynamic Polymorphism, or Late Binding.

Difference between overloading and overriding:

Criteria	Overriding	Overloading
Method signature	Must be same	Must be different.
Return type	Must be same.	Never considered.
Accessibility modifier	Must be same or Less restrictive.	Never considered.
Throws clause	Must not throw new checked exceptions.	Never considered.
Declaration context	A method must only be overridden in a subclass.	A method can be overloaded in the same or in a sub class.
Method call resolution	Method overriding resolved at runtime based on actual object type. It is also called as **dynamic polymorphism** or **late binding**.	Method overloading resolved at compile time based on Reference type. It is also called as **static polymorphism** or **early binding**.

Constructor

The purpose of the constructor is **initialization** followed by **instantiation**.

Constructor Rules:

a) The name of constructor must be same as class name.
b) The only applicable modifiers for the constructors are:
   i) Public ii) protected iii) default modifier iv) private
c) The constructor must not have any return type not even void.

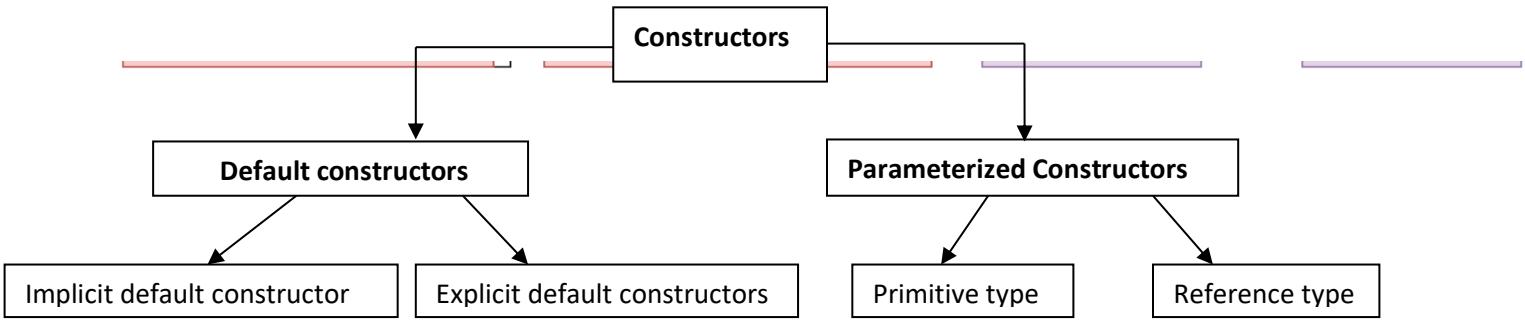
Example:

```
public   class   Box{
         Int   width;
         Int  height;
         Int depth;
         public Box(int w, int h, int d){ //Initialization
```

```
            width = w;
            height = h;
            depth = d;
        }
}
```

Box b = **new Box(1,2,3); //Instantiation**

Constructors are either Default constructors or Parameterized constructors:



Default constructor

Constructor without parameters is called as Default constructor. It is also called as **no-arg constructor**.
There are two types of default constructors:

a) Implicit Default Constructor

If no constructor is declared explicitly inside class, the Java Compiler automatically includes constructor without parameters called as Implicit Default Constructor.

Example:

```
public class Box{}

//After compilation
public class Box{
        public Box(){} //implicit default constructor
}
```

b) Explicit Default Constructor

We can declare constructor without parameters list explicitly is called as explicit default constructor.

Example:

```
Class Box{
        Box(){ //explicit default constrctor
                Width = 1;
                Height = 2;
                Depth = 3;
        }
}
```

Parameterized Constructor

Constructor can take either Primitive or Reference type as its parameters is called as **parameterized constructor**.

Example:

```
class Box{
        int width;
        int height;
        int depth;
        Box(){  //no-arg constructor
                Width =1;
                Height = 2;
                Depth = 3;
        }
        Box(int width){
                this.width = width;
                height = 2;
                depth = 3;
        }
        Box(int width, int height){
                this.width = width;
                this.height = height;
                depth = 3;
        }
        /*Box(int width, int height, int depth){
                this.width = width;
                this.height = height;
                this.depth = depth;
        }*/
        //short cut approach
        Box(int width, int height, int depth){
                this(width, height); //calls matching constructor from the same class
                this.depth = depth;
        }
        Box(Box b){
                this.width = b.width;
                this.height = b.height;
                this.depth = b.depth;
        }
}
public class ConstructorDemo {
        public static void main(String[] args) {
                Box b1 = new Box(1,2,3);
                Box b2 = new Box(b1);
        }
}
```

Constructors are overloaded with each other. Also, Constructors cannot be inherited to subclass.

Constructor Chaining

Super class constructors are always executed before subclass constructors is called as constructor chaining.

Example:

```
class One{
        One(){
                System.out.println("1");
        }
}
class Two extends One{
        Two(){
                System.out.println("2");
        }
}
class Three extends Two{
        Three(){
                System.out.println("3");
        }
}
public class ConstructorChainingDemo {
        public static void main(String[] args) {
                Three obj = new Three();
        }
}
```

O/P:

1
2
3

super or this

There are two forms of 'super' or 'this': i) without parantheses ii) with parantheses

'this' without parentheses are used to invoke **fields** and **methods** from the same class.

'super' without parantheses is used to invoke its immediate superclass **fields** and **methods**.

'this()' is used to call matching constructor from the same class.

'super()' is used to call matching constructor from immediate superclass.

Example:

```
        class One{
                Int age = 20;
        }
        class Two extends One{
                Int age = 30;
                Void disp(){
                        SOP(age); // 30
                        SOP(this.age);//30
                        SOP(super.age); // 20
                }
```

```
}
```

final

The 'final' is a modifier used with:

a) Variables b) Methods c) Classes

final w.r.t variables

The value of the final variable is always ***fixed*** i.e., once it is initialized, it never re-initialized.

Example:

```
final int SIZE = 10;
//SIZE = 20;      //C.E. Final variable cannot be re-initialized.
```

It is always recommended to use Upper case for final variables names.

Final w.r.t methods

Final methods are always fully (completely) implemented.

Example:

```
class Figure{
        final void move(){}
        final void resize(){}
        final void show(){}
        final void hide(){}
        public double area(){
                return 0.0; //Unknow
        }
}
```

Final methods cannot be overridden i.e., final keyword **prevents** method overriding.

final w.r.t classes

If all methods in class are completely implemented, then instead of declaring every method as final, better declare class itself as final.

By default, all methods in a final class are final.

Example:

```
public final class One{
        public void meth1(){}
        public void meth2(){}
}
```

Final classes cannot be extended.

Example:

```
class Two extends One{}//C.E
```

Note:

1) Final with variables prevents re-initialization.
2) Final with methods prevents method overriding.
3) Final with classes prevents inheritance.

static

The 'static' modifier is used with:

a) Variables b) methods c) Blocks d) Inner class

Static members are **Independent** of any object creation i.e., static members are **Common** across all objects. Static members never associated with any object but directly associated with class. Hence, static variables are also called as **Class variables or Global variables**.

Example:

```
class Box{
        Private int width;
        Private int height;
        Private int depth;
        public static int count; //static variable
        Box(int w, int h, int d){width = w; height = h; depth = d; count++;}
}
Box b1 = new Box(1,2,3);
Box b2 = new Box(4,5,6);
sop(Box.count); //2
```

Static members are executed when class is loaded into JVM. Hence, static members are accessed without (or before) creating an object itself.

Note:

1) Instance variables are associated with **Heap area**.
2) Static variables are associated with **Method area**.
3) Local variables are associated with **Stack area**.
4) Final variables are associated with **Constant Pool area**.

Static methods

Rules:

1) A static method cannot access instance variables.
2) A static method cannot access instance methods.
3) Cannot use this or super from static context.

We can use static members from non-static context, but not vice versa.

It is always recommended to use class name to access static members rather than object name.

abstract

The **'abstract'** modifier is used with:

a) methods b)classes.

Declare method as abstract if the method implementation is **unknown**. Always, abstract methods must ends with **semi-colon**.

Example:

```
public abstract double area();
```

Abstract class Rules:

Declare class as abstract if:

a) Atleast one abstract method, but all other methods are concrete.
b) All methods are abstract methods, i.e., none of the method is a concrete method.
c) All methods are concrete, i.e., none of the method is an abstract method.

If a class inherits abstract class using extends keyword, the subclass must either implement all abstract methods or declare subclass also as abstract class.

Abstract classes must not be instantiated, but abstract classes are used as reference variables to achieve dynamic polymorphism.

Example:

```
public abstract class Figure{
        Int width;
        Int height;
        Figure(int width, int height){this.width = width;  this.height = height;}
        Public final void move(){}
        Public final void resize(){}
        Public final void hide(){}
        Public final void show(){}

        public abstract double area();
}
class Rectangle extends Figure{
        Rectangle(int w, int h){super(w, h);}
        @Override
        public double area(){
                Return width * height;
        }
}
Class Triangle extends Figure{
        Triangle(int w, int h){super(w, h);}
        @Override
        Public double area(){
                Return 0.5 * width * height;
        }
}

Public class AbstractDemo{
        Public static void main(String[] args){
                //Figure fRef = new Figure(1,2); //C.E since abstract classes cannot be instantiated.
                Figure fRef = new Rectangle(10,20);//Subclass object is assigned to superclass ref variable
                SOP(fRef.area());          // The area() method from rectangle is invoked.
                fRef = new Triangle(10,20);
                SOP(fRef.area());          // The area() method from triangle is invoked.
        }
}
```

Advantages with Abstract classes:

1) Forces the subclass to provide method implementation
2) Achieves Dynamic Polymorphism.

Interfaces

Interface provides service details but not its implementation details.

For example, the stack interface says its operations push() and pop() but not its actual implementation details such as array or linked list. We (service provider) can change stack implementation from array to linked list without affecting to client applications (service consumer).

Service Consumer

Service Details (interface)

Service Provider

Syntax:

<modifier> **interface** <interface name> **[extends** interface**1,**… interface**n]** {
    [field declarations]
    [method declarations]
}

Fields w.r.t Interface

By default, all interface fields are **public static final**.

Hence, All of the following field declarations are equal:

a) **public static final** int SIZE = 10; //Declaring public static final explicitly is redundant but not error.
b) static final int SIZE = 10;
c) final int SIZE = 10;
d) int SIZE = 10;

All interface variables must be initialized in the declaration section.

Example:

Public static final int SIZE = **10**; //declaration cum initialization
Public static final int SIZE; //C.E saying Missing Initialization

Methods w.r.t Interface

By default, all interface methods are **public abstract i.e.,** interface is a pure abstract class.

Hence, All of the following methods declarations are equal:

a) **public abstract** void meth(); //Declaring public abstract explicitly is redundant but not error.
b) public void meth();
c) abstract void meth();
d) void meth();

All methods in an interface are abstract, hence, interfaces cannot be instantiated. However, interfaces can be used as reference variables.

Example:

```
public interface Stack{
        public void push(int x);
        public int pop();
}

public class StackImpl implements Stack{
        private final int SIZE = 10;
        @Override
        public void push(int x){
                //impl using arrays
        }
        @Override
        public int pop(){
                //impl using arrays
        }
}

Stack sRef = new StackImpl(); //Ok
sRef.push(10);
sRef.pop();
```

Constructors w.r.t interface

All variables in an interface are final constants, which must be initialized in the declaration section itself.
Also, all methods in an interface are abstract, such an interfaces cannot be instantiated.
Hence, Interface does not need constructors at all i.e., **interfaces never have constructors at all**.

Inheritance w.r.t Interface

A class inherits another class using **extends** keyword.
A class inherits interface(s) using **implements** keyword.
An interface inherits interface(s) using **extends** keyword.
Either class or an interface can inherit any number of interfaces.

Example:

```
Public interface One{}
Public class Two
implements One{}
```

```
Public interface One{}
Public interface Two
extends One{}
```

```
Public interface One{}
Public interface Two{}
Public class Three
implements One, Two{}
```

```
Public interface One{}
Public interface Two{}
Public interface Three
extends One, Two{}
```

Java supports multiple inheritance by inheriting multiple interfaces.

The subclass of an interface either implement all abstract methods or declare subclass as an abstract class.
The accessibility modifier of the subclass method must be always **public**.

Example:

```
        Public Interface One{
                public void meth1();
```

```
        Public void meth2();
}
```

class Two implements One{
public void meth1(){}
public void meth2(){}
}
	abstract class Two implements One{
	public void meth1(){}
	}

Marker Interface

If our class gets special privilege by inheriting an interface, such an interface is called as marker interface.

Example:

Java.io.Serializable
Java.lang.Runnable
…

If an interface does not have any methods, it is always called as Marker interface.

Example:

Java.io.Serializable

Even though an interface contains methods which gives special privilege to our class, then such an interface is also called as Marker Interface.

Example:

Java.lang.Runnable - run() method.

Question: Though interface is a pure abstract class, why do we need an interface?

To support multiple inheritance and if a subclass already implements another class, then subclass cannot inherit our abstract class. This restriction is not applicable for an interface, since a class can inherit any number of interfaces.

Difference between abstract class and Interface:

Abstract class	Interface
Can contain concrete methods.	All methods are abstract.
Can contains non final & non static variables.	All variables must be static and final.
Can contains any number of constructors.	Never contains constructors at all.
Does not support multiple inheritance.	**Supports multiple inheritance.**