

Java and OOPs

pratibha.malhotra@sun.com

Agenda

- Structured Versus Object Oriented Programming
- OOP Concepts
- Java Fundamentals
- OOP in Java
- Comparison b/w Java and other languages
- Q & A
- Exercise

Structured Programming

- In structured programming, the data and the operations on the data are separate. A data structure is created hold attributes and is sent to functions to be operated on.
 - > Link between functions and data is function parameter.
 - > In an event of changes to MyCar, coder has to search for all places where data structure is being used and has to be modified.

```
Public class MyCar  
{ int numDoors;  
  Float speed;  
}  
public void speedUp(Mycar m)  
{.....}  
public void speedDown(Mycar m)  
{.....}
```

Object Oriented Programming

- Object-oriented programming has been around since 1967. It really came to the forefront of programming paradigms in the mid-1980s however.
- In Object-oriented programming, data and operations are part of the same entity. This more closely models the real world, in which all objects have both attributes and code associated with them.
 - > There is no need to pass in a reference to a Data structure as Methods implicitly know about the variables of their class and have full access to them.

```
Public class MyCar  
{  
  int numDoors  
  float speed;  
  public void speedUp()  
  {.....}  
  public void speedDown()  
  {.....}  
}
```

Structured Versus Object Oriented Programming

- In world of structured programming
 - > Data lies separate from code.
 - > No data abstraction or info hiding.
 - > Programmer is responsible for organizing everything in to logical units of code/data.
 - > No help from compiler/language for enforcing modularity.
- OOP comes to the rescue
 - > Model everything as objects.
 - > Keep data near the relevant code.
 - > Avoid reinventing the wheel.
 - > Easier to understand, manage and maintain.
 - > Simplify testing and debugging.

Why OOP?

- Promotes and facilitates software reuse.
 - > Save development time and cost
- Enables Easy Debugging
 - > Classes can be tested independently
 - > reused objects have already been tested
- Facilitates interoperability
- Results in software that is easy to modify, extend and maintain
- Reduces integration effort

Object Oriented Design Principle

- Identify interacting Objects.
- Characterize each object, establish attributes.
- Identify the data and operations within each object.
- Identify requests answered by each object.
- Identify services required of other objects.
- Establish relationships to other objects.
- Group similar objects together.
- Implement common super classes.
- Implement different objects as classes.

Pillars of OOPS

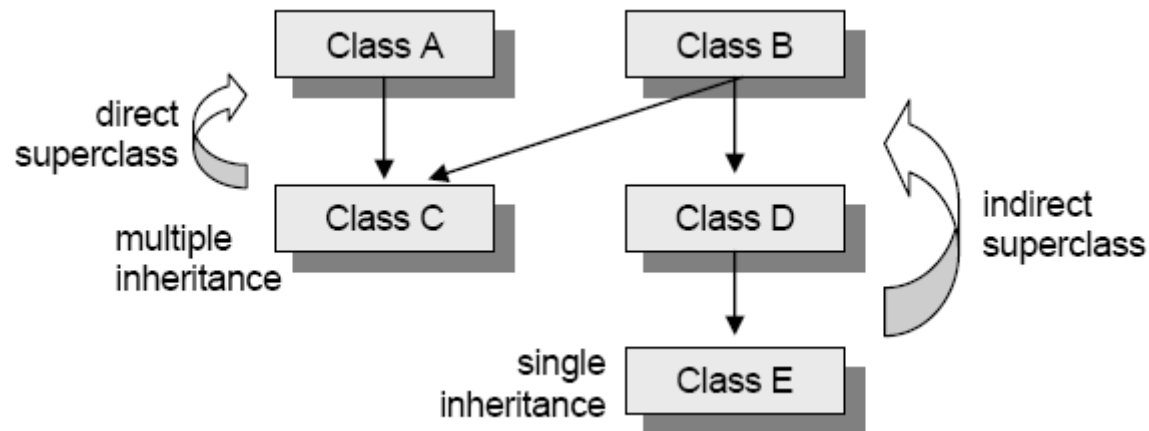
- Inheritance
- Encapsulation
- Polymorphism

Inheritance

- It is the capability of a class to use the properties and methods of another class while adding its own functionality
- Represents is-a relationship
- **Analogy**
Just as inheritance in real life (you get traits from your parent), OOP inheritance is what you get from your parent (class)
- **Why?**
 - > Enhances ability to reuse code.
 - > Make designing much simpler and cleaner process.
- **Advantage**
 - > Efficiency, extensibility

Inheritance - cont.

- Types of Inheritance
 - > Single Inheritance
 - > Multiple Inheritance
 - > Multilevel Inheritance



Encapsulation

- It is the ability of an object to place a boundary around its properties (data, methods).
- It seals the data and methods safely inside the 'capsule' of a class, where it can be accessed only by trusted users (ie methods of the class).
- **Analogy**
 - > While driving a car, user don't have to know the details of cylinders, ignition etc.
- **Why?**
 - > No need to know implementation details of a component to use it.
 - > Implementation can change without effecting any calling code.
- **Advantage**
 - > Provides Modularity, Code quality, ease of maintenance

Polymorphism

- It is a greek and means “many forms”.
- Same word or phrase can mean different things in different contexts
- **Analogy**
 - > Students response to a school bell.
 - > In English vocab, bank can mean side of a river or place to put money.
- **Why?**
 - > To design, produce and describe software so that it can be easily used without knowing the details of how it works.
- **Advantage**
 - > Power!!

Class, Object ?

- Class
 - > Class is described as a blueprint for building objects.
 - > It specifies properties and methods, object is associated with.
- Object
 - > Objects are specific and concrete, come into existence at some specific time, persist for some duration and then disappear.
 - > Each object has it's own memory for maintaining state(fields)

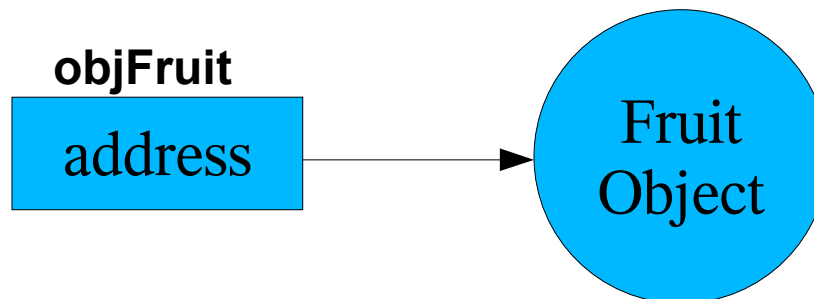
Instantiation ?

- It is the process by which individual objects are created.

Class objectReference = new Constructor();

- > Fruit objFruit; // Declaration
- > objFruit = new Fruit(); // Instantiation
- > Fruit objFruit = new Fruit(); // Declaration and Instantiation

- new operator allocates memory for an object.



Class in Java

- Java Class

```
[public] [abstract] [final] class class_name
[extends super_class_name] [implements interface_name]
{
  constructor declarations;
  attribute declarations;
  method declarations;
}
```

- Members of a Java Class

- > Constructors
- > Attributes (Static and Instance)
- > Methods (Static and Instance)
- > Inner classes (Java 1.1 and later)

Members of a Java Class

- Constructor

- > Method which is invoked when an object of a class is created.
- > It has no return type.
- > Its name must be same as the name of class.

```
[public/private/protected] constructor_name(parameter_list)  
{  
}
```

- > It can accept parameter.
- > If no constructor is defined, default constructor is automatically used.
- > Unlike methods, a constructor cannot be abstract, static, final or synchronized.

Members of a Java Class

- Method
 - > Class behaviors are represented in Java by methods.

```
[public/private/protected] [final] [static/abstract/native]  
return_datatype method_name(parameter_list)  
{  
}
```

Members of a Java Class - cont.

- **Static/Class Members**

Static members are associated with a class, rather than with an instance.

A Static attribute/method is essentially a **global attribute/method**.

- > Static Block
- > Static Attribute
- > Static Method

```
public static double PI = 3.14;
```

Inside the class, it can be referred as PI

Outside the class, it will be referred as Circle.PI

Note : For clarity, It is better to specify class name even in the same class.

Try - new Circle().PI

static

- Any method that is independent of instance state is a candidate for being declared as static.

```
class CreateObject
{
    static int instantiations = 0;
    public CreateObject()
        { instantiations++; }
    public static int instances()
        { return instantiations;}
    public static void main(String s[])
        { new CreateObject();
          System.out.println(CreateObject.instantiations );
        }
}
```

- Compiler produce more efficient code because no implicit object parameter has to be passed to the method
- Example - java.util.Math, Java.util.Random

static - cont.

```
class A
{
    static{ System.out.println("I am in static block");
        }
    A()
        {System.out.println("I am in constructor");
        }
    static int val = 10;
    static void callme()
        { System.out.println("I am in static method, val "+val);
        }
}

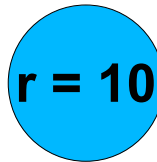
class TestStatic
{
    public static void main(String[] args)
        { A.callme();
        A obj = new A();
        obj.callme();
        }
}
```

Members of a Java Class - cont.

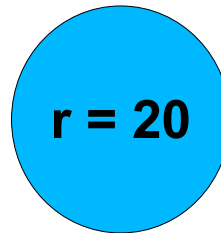
- Instance Members

Any member declared without static modifier is an instance member.

```
Circle c = new Circle();  
c.radius = 10;
```



```
Circle c1 = new Circle();  
c1.radius = 20;
```



Every object has its own copy of Instance members.

- > Instance Attribute
- > Instance Method

Members of a Java Class - cont.

- **Inner Classes**

- > Inner classes are classes nested inside another class.
- > Have access to outer class attributes/methods, **even if marked as private**

```
public class InnerClassTest
{
    private static String hiding = "I am Private Attribute, u can't see me !!";

    public static class InnerClass
    {
        public static void showMe()
        {
            System.out.println(hiding);
        }
    }

    public static void main (String args[])
    {
        InnerClass.showMe();
    }
}
```

Sample Java Class

```
class MyClass
```

```
{
    MyClass(int i)
        {myInstanceVar = i;
        }
    static int myStaticVar = 100;
    public static void myStaticMethod()
        {System.out.println("I am a static method");
        }
    int myInstanceVar;
    public void myInstanceMethod()
        {System.out.println("I am an Instance method");
        }
    public static void main(String s[])
    {
        myStaticMethod();
        System.out.println("myStaticVar - "+myStaticVar);
        MyClass objMyClass = new MyClass(1000);
        objMyClass.myInstanceMethod();
        System.out.println("myInstanceVar - "+objMyClass.myInstanceVar);
    }
}
```

Constructor

Static Member

Instance Member

Inheritance in Java

- Java uses the extends keyword to set the relationship between a child class and a parent class.
- Except Object, which has no superclass, every class has one and only one direct superclass.
- Multilevel Inheritance
- Multiple Inheritance is not supported.

Inheritance cont.

- **Superclass:**
 - a *parent* or *base* class. Superclass wrongly suggests that the parent class has more functionality than the subclass. Generally a subclass is a more specialised form of the superclass.
- **Subclass:**
 - a *child* class that inherits from, or extends, a superclass.
- Apple extends Fruit
 - > `Fruit objFruit = new Apple();`
 - > Apple can be used wherever Fruit objects are called for. Reverse is not necessarily valid.

Encapsulation in Java

- It can be achieved by declaring attributes and let coder access it using public getter/setter methods in the class.

```
class Encapsulation
{   private int val;
    public void calSquare()
        { val = val*val;}

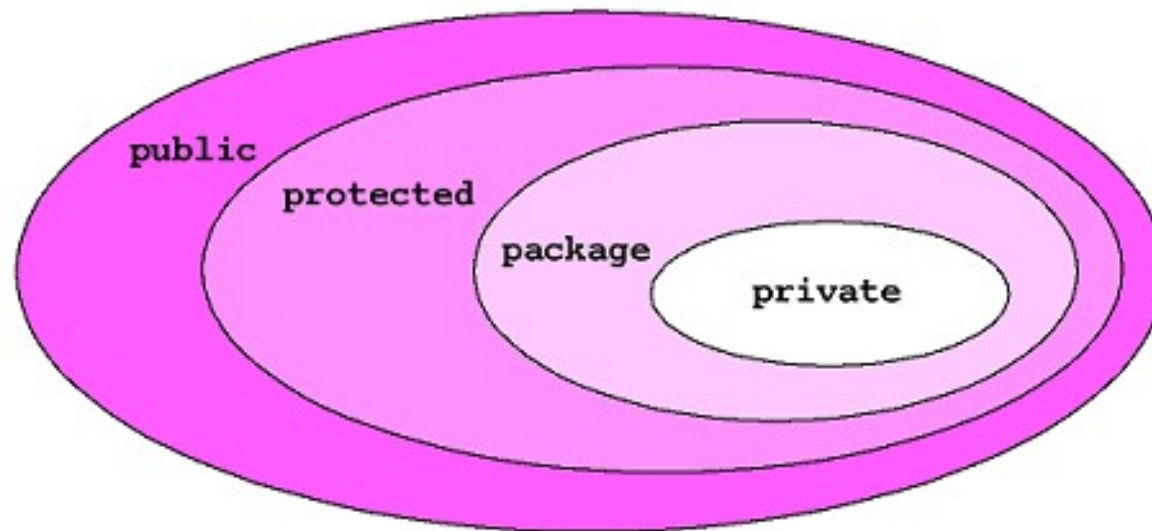
    public int getVal()
        { return val;}

    public void setVal(int val)
    { this.val = val;}
}

public class main
{ public static void main(String s[])
    { Encapsulation obj = new Encapsulation();
      obj.setVal(10);
      obj.calSquare();
      System.out.println(obj.getVal());
    }
}
```

Access Modifiers

- In Java, encapsulation is implemented using Access Modifiers.
- Determine whether a method or a data variable can be accessed by another method in another class.



Access Modifiers - cont.

External access	public	protected	(default) package	private
Same package	yes	yes	yes	no
Derived class in another package	yes	yes (inheritance only)	no	no
User code	yes	no	no	no

Abstract Class

- An *abstract class* is a class that is declared using keyword ***abstract***.
- It may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be subclassed.
- An *abstract method* is a method that is declared without an implementation
 - > **abstract void moveTo(double deltaX, double deltaY);**

Abstract Class - cont.

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

- When an abstract class is subclassed, the subclass usually provides implementations for all abstract methods in parent class. However, if it does not, the subclass must also be declared *abstract*.

Interface

- If an abstract class contains *only* abstract method declarations, it should be declared as an interface instead.

```
[public ] [ abstract ] interface interface_name  
  {  
    // constants and method signature declarations  
  }
```
- Provide a workaround to Java's lack of support for multiple inheritance.
- All of the methods in an *interface* are implicitly abstract, hence there is no need to use abstract modifier with interface methods.
- Just as classes extend other classes, interfaces can extend other interfaces (but not classes).

```
extends interface_name [ ',' interface_name ... ]
```
- Marker Interfaces (Empty Interfaces)
 - > examples include Cloneable, Serializable,

Polymorphism in Java

- It allows Java programs to be written more abstractly. More abstraction allows more efficiency and less redundancy.
- It allows inheritance and interfaces to be used more abstractly.
- Same method can do different things, depending on the class that implements it.
- Method invoked is associated with OBJECT and not with reference.

- Types of Polymorphism -
 - > Method Overloading.
 - > Method Overriding.

Overloading

- Acc to JavaLangSpec-3.0,
two methods are said to be overloaded if both are declared in the
 - > same interface/class or
 - > both are inherited by an interface/class or
 - > one declared and one inheritedhave the same method name but different signatures.
- method selection is done during compilation
- Unlike Overriding, It never of itself results in a compile-time error.
- Unlike Overriding, there is no required relationship between the return types or throws clauses of overloaded methods.

Overloading - cont.

```
class SuperClass
{ public void move(String str)
  { System.out.println("Super");
  }
}

class SubClass extends SuperClass
{ public void move(int point)
  { System.out.println("Sub");
  }

public static void main(String s[])
  { SubClass obj = new SubClass();
    obj.move(5);
    SuperClass obj1 = new SuperClass();
    obj1.move("java");
  }
}
```

Overriding

- An instance method `m1` declared in a class `B` overrides another instance method, `m2`, declared in class `A` iff all of the following are true:
 - > `B` is a subclass of `A`.
 - > Either, `m2` is public, protected or declared with default access in the same package as `B`


Requirements in Overriding

- If a method declaration $d1$ with return type $R1$ overrides the declaration of another method $d2$ with **return type** $R2$, then $d1$ must be return-type substitutable for $d2$, or a compile-time error occurs.
 - > Furthermore, if $R1$ is not a subtype of $R2$, an unchecked warning must be issued.

```
class A
{ void callme()
    {System.out.println("A.callme()");
    }
}
```

```
class B extends A
```

```
{ int callme()
    {System.out.println("B.callme()");
    }
}
```



- A method declaration must not have a **throws clause** that conflicts with that of any method that it overrides; otherwise, a compile-time error

Requirements in Overriding - cont.



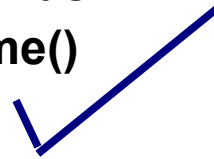
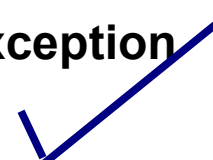
- A method declaration must not have a throws clause that conflicts with that of any method that it overrides. Otherwise, a compile-time error.
 - > Overriding method must only throw an Exception that is more restrictive than overridden method.
 - > Overriding method might decide not to throw any Exception.
 - > Overriding method can throw RuntimeException & its subclasses.

Requirements in Overriding - cont.

```

class A
{ void callme() throws IOException
  {...}
}
class B extends A
{ void callme() throws Exception
  {...}
}
class C extends A
{ void callme() throws FileNotFoundException
  {...}
}
class D extends A
{ void callme()
  {...}
}
class E extends A
{ void callme() throws RuntimeException
  {...}
}

```

super

- super keyword can be used for accessing following members of superclass
 - > overridden method.
 - > hidden attribute.
 - > constructor .

super - cont.

```
class Superclass1
```

```
{    Superclass1(int val)
    { System.out.println("I am in superclass's Constructor, val "+val); }
    String name = "java";
    public void printMethod()
    { System.out.println("I am in Superclass's printmethod()"); }
}
```

```
public class Subclass1 extends Superclass1
```

```
{    Subclass1(int val)
    { super(val*2);
      System.out.println("I am in subclass's Constructor, val "+val);
    }
    String name = "C++";
    public void printMethod()
    { super.printMethod();
      System.out.println("I am in Subclass's printmethod(), name - "+name);
      System.out.println("name in super class - "+super.name);
    }
    public static void main(String[] args) {
        Subclass1 obj = new Subclass1(5);
        obj.printMethod();
    }
}
```

this

- **this** refers to the current instance of the class.

```
public class Area
{
    public int value;

    public int calArea(int value)
    {
        this.value = value * value;
        return this.value;
    }

    public static void main(String s[])
    {
        Area objArea = new Area();
        System.out.println("before "+objArea.value);
        objArea.calArea(5);
        System.out.println("after "+objArea.value);
    }
}
```

Importance of super, this

class Parent

```
{ Parent()
  { System.out.println("Parent() constructor"); }
}
```

class Child extends Parent

```
Step 3 { Child()
        { this(25);
          System.out.println("Child() constructor");
        }
        Child(int x)
        { super();
          System.out.println("Child(" + x + ") constructor");
        }
    }
```

Step 2

Step 1

class ConstructorChain

```
{ public static void main(String[] args)
  { Child c = new Child();
  }
}
```

Importance of super, this - cont.

- The first line of every constructor must be either
 - > A **this** call to another constructor in the same class.
 - > A **super** call to a parent constructor.
- If no constructor call is written as the first line of a constructor, the compiler automatically inserts a call to the **parameterless** constructor of Superclass.

Hiding static methods

```
class Super
```

```
{    static String greeting() { return "Goodnight"; }  
    String name() { return "Tom"; }  
}
```

```
class Sub extends Super
```

```
{    static String greeting() { return "Good Morning"; }  
    String name() { return "Dick"; }  
}
```

```
class TestStaticMethods
```

```
{    public static void main(String[] args)  
    {    Super s = new Sub();  
        System.out.println(s.greeting() + ", " + s.name());  
    }  
}
```

Solution

output:

Goodnight, Dick

- > Super s = new Sub();
- > Resolution regarding static method invocation depends on the type of object-reference, namely Super.
- > Resolution regarding instance method invocation depends on object, object-ref is pointing to namely Sub.
- > Resolution regarding static-method invocation happen at compile time, whereas at run time for instance methods.

Overriding versus Hiding

	SuperClass Instance Method	SuperClass Static Method
Subclass Instance Method	Overrides, except private	Compile time Error
Subclass Static Method	Compile time Error	Hides

	SuperClass Instance Attribute	SuperClass Static Attribute
Subclass Instance Attribute	Hides	Hides
Subclass Static Attribute	Hides	Hides

```

public class SuperClass
{ public static String callMe()
    {.....}
}
public class SubClass extends SuperClass
{ public static String callMe()
    {.....}
}

```

Dynamic Method Dispatch

```
class SuperClass
{ int value = 100;
  void callMe()
    { System.out.println("I am in Super Class"); }
}















class SubClass extends SuperClass
{ int value = 10;
  void callMe()
    { System.out.println("I am in Sub Class"); }
}

public class TestDMD
{
  public static void main(String s[])
  {
    SuperClass objSuperClass = new SubClass();
    objSuperClass.callMe();
    System.out.println(objSuperClass.value);
  }
}
```

Dynamic Method Dispatch

- superclass = subclass
always valid
- subclass=(subclass)superclass
valid at compile time, checked at runtime. if is invalid
then the exception **ClassCastException** is thrown.
- subclass = superclass
not valid at compile time, needs a cast
- someClass = someUnrelatedClass
not valid, won't compile
- somcClass = (someClass)someUnrelatedClass
not valid, won't compile

Java versus other languages

	<i>Java</i>	<i>SmallTalk</i>	<i>TCL</i>	<i>Perl</i>	<i>Shells</i>	<i>C</i>	<i>C++</i>
<i>Simple</i>							
<i>Object Oriented</i>							



Feature exists



Feature somewhat exists



Feature doesn't exist

Q and A

Exercise

- 1) Can one access instance attributes/methods in a static method?
- 2) Can one access static attributes/methods in a instance method?
- 3) Can one access static member of any other class in a static method?
- 4) Can an instance method override static method?
- 5) Can an instance attribute hide static attribute?

Exercise - cont.

- 1) When should an abstract Class Implement an Interface?
- 2) Can an abstract class have static attributes and static methods?
- 3) When does one choose an abstract class over an interface?
- 4) Is following code valid?

```
class A  
{   int x = 0;  
    int getX(){return x};  
}  
class B extends A  
{   float x = 0.0f;  
    float getX(){return x};  
}
```

Exercise - cont.

Given the following, write the java code for the following, compile and run it.

Step 1:- There is a class Vehicle. It has a public method start().

write down the two subclasses for this vehicle in same package as

a:- Class Car, with public method driving()

b:- Class Aircraft, with public method flying().

you Provide some messages in all the methods so that when they are called get the desired output printed.

Now, write down one more class say MyApplication to instantiate these classes.

Compile and run it using Netbeans.

Step 2:- Now try the multilevel Inheritance by creating subclasses for Aircraft class in same package as

a:- Class Jet, with public method zoom().

b:- Class Whirlbird, with public method whirl().

you Provide some messages in all the methods so that when they are called get the desired output printed.

Now, try to instantiate these new classes by changing the class MyApplication.

Compile and run it.

Exercise - cont.

- Step 3:-** Check, what happens if we change the access modifier of start() in class Vehicle from public to private.
First think, and explain the reason, then try it out by Compiling and running it.
- Step 4:-** Check, what happens if we change the access modifier of method start() in Vehicle class to protected.
Think and explain the reason. Try it out by Compiling and running it using Netbeans.
- Step 5:-** Now try to override the start() of Vehicle Class in any of the subclass, say Class Car.
Print the output of Car class start() method.
What changes you have to do, to call the start() of Vehicle class from Car class?
First think then try, explain the reason.

Bibiliography

- <http://java.sun.com>
- Javadocs jdk 1.5
- Java Developers' Almanac
- <http://www.javapassion.com>

If an exact signature is not possible, one closest via widening is used. Widening means that values of smaller types are cast into values of larger types. eg. int to long, int to float, float to double