Chapter 2

Java Architecture and OOPs

2.1 Java Architecture and OOPs

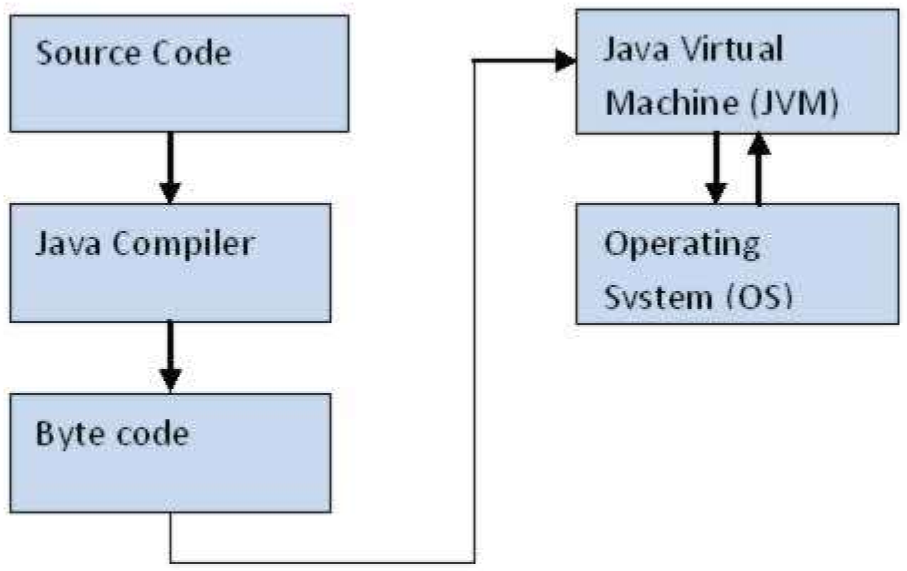
Java Architecture and Object Oriented Programming is explained in the section below.

2.1.1 Introduction

Java is a general-purpose computer-programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture. As of 2016, Java is one of the most popular programming languages in use, particularly for client-server web applications, with a reported 9 million developers. Java was originally developed by James Gosling at Sun Microsystems (which has since been acquired by Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The latest version is Java 10, released on March 20, 2018, which follows Java 9 after only six months.

2.1.2 Compilation Process

Java combines both the approaches of compilation and interpretation. First, java compiler compiles the source code into byte code. At the run time, Java Virtual Machine (JVM) converts this byte code and generates machine code which will be directly executed by the machine in which java program runs.

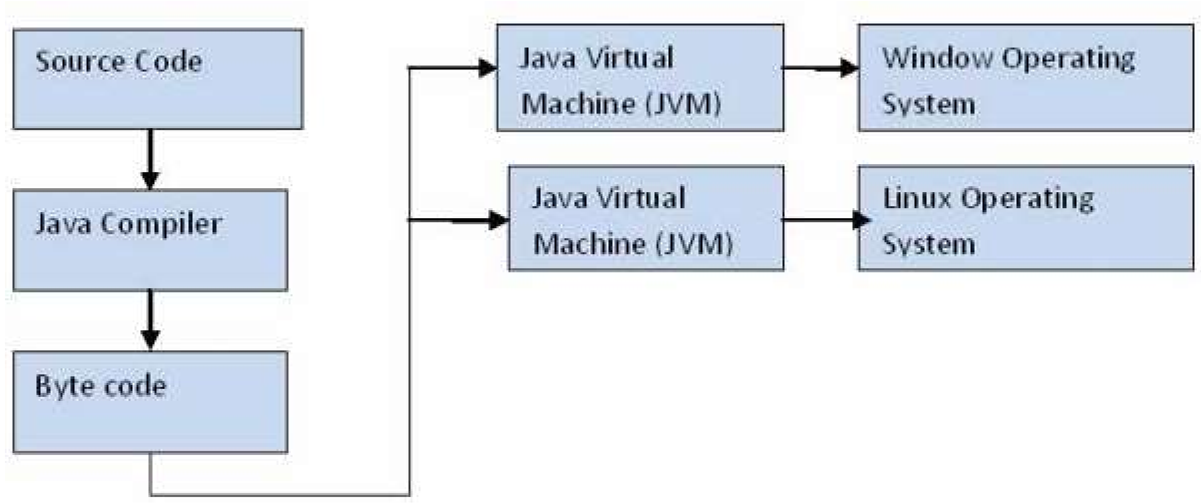


2.1.3 JVM (Java Virtual Machine)

JVM is a component which provides an environment for running Java programs. JVM converts the byte code into machine code which will be executed the machine in which the Java program runs.

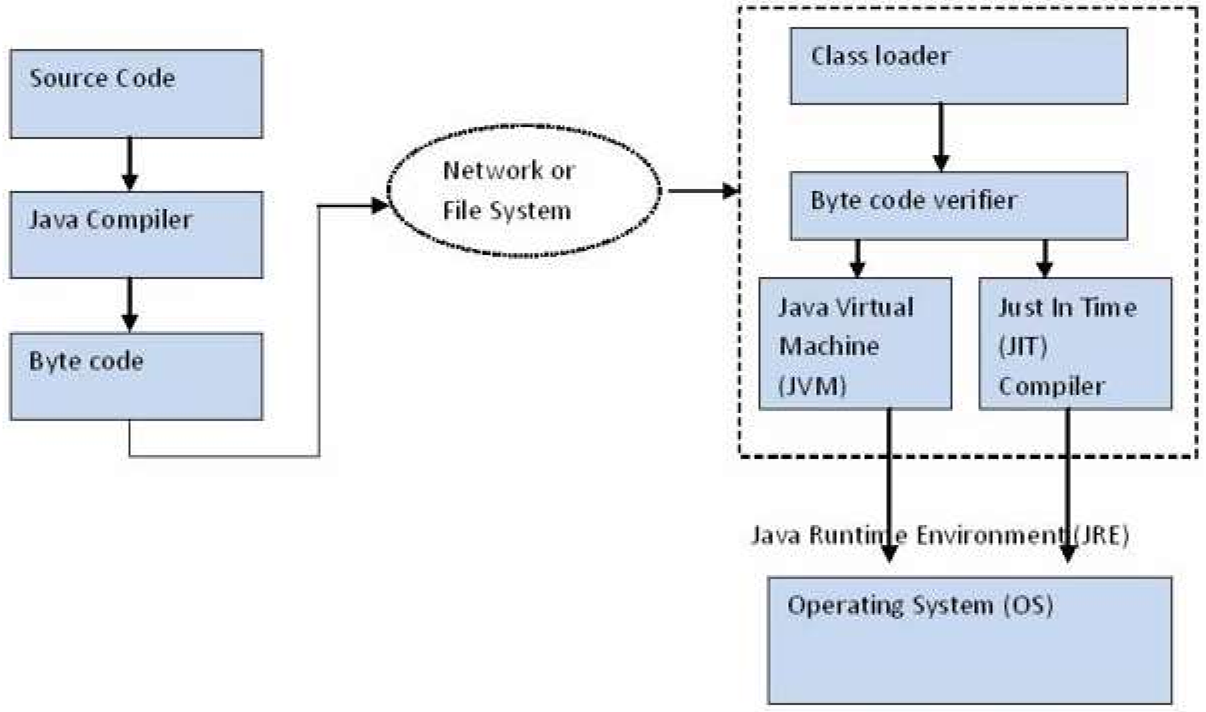
2.1.4 Why Java is Platform Independence?

Platform independence is one of the main advantages of Java. In another words, java is portable because the same java program can be executed in multiple platforms without making any changes in the source code. You just need to write the java code for one platform and the same program will run in any platforms. But how does Java make this possible?
First the Java code is compiled by the Java compiler and generates the byte code. This byte code will be stored in class files. Java Virtual Machine (JVM) is unique for each platform. Though JVM is unique for each platform, all response the same byte code and convert it into machine code required for its own platform and this machine code will be directly executed by the machine in which java program runs. This makes Java platform independent and portable.



2.1.5 JRE (Java Runtime Environment)

A Java virtual machine (JVM) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages and compiled to Java bytecode. The JVM is detailed by a specification that formally describes what is required of a JVM implementation. Having a specification ensures interoperability of Java programs across different implementations so that program authors using the Java Development Kit (JDK) need not worry about idiosyncrasies of the underlying hardware platform.



2.1.6 Class Loader

Class loader loads all the class files required to execute the program. Class loader makes the program secure by separating the namespace for the classes obtained through the network from the classes available locally. Once the bytecode is loaded successfully, then next step is byte code verification by bytecode verifier.

2.1.7 Bytecode Verifier

The bytecode verifier verifies the byte code to see if any security problems are there in the code. It checks the byte code and ensures the followings. 1. The code follows JVM specifications. 2. There is no unauthorized access to memory. 3. The code does not cause any stack overflows. 4. There are no illegal data conversions in the code such as float to object references. Once this code is verified and proven that there is no security issues with the code, JVM will convert the byte code into machine code which will be directly executed by the machine in which the Java program runs.

2.1.8 JIT (Just in Time Compilation)

When the Java program is executed, the byte code is executed by JVM. But this interpretation is a slower process. To overcome this difficulty, JRE include the component JIT compiler. JIT makes the execution faster. If the JIT Compiler library exists, when a particular bytecode is executed first time, JIT complier compiles it into native machine code which can be directly executed by the machine in which the Java program runs. Once the byte code is recompiled by JIT compiler, the execution time needed will be much lesser. This compilation happens when the byte code is about to be executed and hence the name "Just in Time". Once the bytecode is compiled into that particular machine code, it is cached by the JIT compiler and will be reused for the future needs. Hence the main performance improvement by using JIT compiler can be seen when the same code is executed again and again because JIT make use of the machine code which is cached and stored.

2.1.9 Why Java is Secure?

As you have noticed in the prior session "Java Runtime Environment (JRE) and Java Architecture in Detail", the byte code is inspected carefully before execution by Java Runtime Environment (JRE). This is mainly done by the "Class loader" and "Byte code verifier". Hence a high level of security is achieved.

2.1.10 Garbage Collecction

Garbage collection is a process by which Java achieves better memory management. Whenever an object is created, there will be some memory allocated for this object. This memory will remain as allocated until there are some references to this object. When there is no reference to this object, Java will assume that this object is not used anymore. When garbage collection process happens, these objects will be destroyed and memory will be reclaimed.

2.2 Java Class and Objects

Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of objectoriented programming. Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A distinguishing feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated. An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of intangible object is banking system. An object has three characteristics:

- State: represents data (value) of an object.
- Behavior: represents the behavior (functionality) of an object such as deposit, withdraw etc.
- Identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior. Each object is said to be an instance of a particular class (for example, an object with its name field set to "Mary" might be an instance of class Employee). Procedures in object-oriented programming are known as methods, variables are also known as fields, members, attributes, or properties. This leads to the following terms:

- Class variables - belong to the class as a whole; there is only one copy of each one
- Instance variables or attributes - data that belongs to individual objects; every object has its own copy of each one
- Member variables - refers to both the class and instance variables that are defined by a particular class
- Class methods - belong to the class as a whole and have access only to class variables and inputs from the procedure call
- Instance methods - belong to individual objects, and have access to instance variables for the specific object they are called on, inputs, and class variables

Objects are accessed somewhat like variables with complex internal structure, and in many languages are effectively pointers, serving as actual references to an single instance of said object in memory within a heap or stack. They provide a layer of abstraction which can be used to separate internal from external code. External code can use an object by calling a specific instance method with a certain set of input parameters, read an instance variable, or write to an instance variable. Objects are created by calling a special type of method in the class known as a constructor. A program may create many instances of the same class as it runs, which operate independently. This is an easy way for the same procedures to be used on different sets of data.
Object-oriented programming that uses classes is sometimes called class-based programming, while prototype based programming does not typically use classes. As a result, a significantly different yet analogous terminology is used to define the concepts of object and instance.

2.3 Class Methods and instances

Data is encapsulated in a class by declaring variables inside the class declaration. Variables declared in these scopes are known as instance variables. Instance variables are declared in the same way as local variables except that, they are declared outside any particular method. Class variables are global to class and to all the instances of class. They are useful in communicating between different objects of the same class for keeping track of global states. Methods are functions that operate on instances of classes in which they are defined. Objects can communicate with each other using methods and can call methods in other classes. Method definition has four parts:

- Name of the method
- Type of object
- List of parameters
- Body of method

The methods which apply and operate on an instance are called instance methods and the methods which apply and operate on a class are called class methods.
Class methods, like class variables, are available to instances of the class and can be made available to other classes. Class methods can be used anywhere regardless of whether an instance of the class exists or not. Methods that provide some general utility but do not directly affect an instance of the class are declared as class methods.

2.4 Inheritance and polymorphism in Java

2.4.1 Inheritance

The process of deriving a new class from an old one is called inheritance or derivation. The old class is referred to as the base class or super class or parent class and new one is called the derived class or subclass or child class. The derived class inherits some or all of the properties from the base class. A class can inherit properties from more than one class or from more than one level.

The main purpose of derivation or inheritance is reusability. Java strongly supports the concept of reusability. The Java classes can be reused in several ways. Once a class has been written and tested, it can be used by another by another class by inheriting the properties of parent class. **Defining a subclass** A subclass can be defined as follows:

```
class SubClassName extends SuperClassName{
       // instance variables

       //Methods
}
```

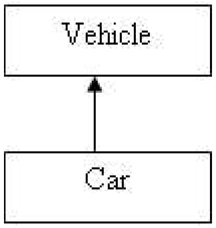
The keyword extends signifies that the properties of the SuperClassName are extended to the SubClassName. The subclass will now contain its own variables and methods as well those of the super class. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it.

2.4.2 Types of Inheritance

The following kinds of inheritance are there in Java.

- Simple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

In java programming, Multiple and Hybrid inheritance is not directly supported. It is supported indirectly by using interface. **Simple Inheritance** When a subclass is derived simply from it's parent class then this mechanism is known as simple inheritance or (a parent class with only one child class is known as single inheritance). In case of simple inheritance there is only one subclass and it's parent class. It is also called single inheritance or one level inheritance.



Pictorial Representation of Simple Inheritance For Example,

```
class Parent
{
        int x;
        int y;
        void get(int p, int q)
        {
                x=p; y=q;
        }
        void Show()
        {
                System.out.println("x=" +x + " and y= " +y);
        }
}
class Child extends Parent
{
        public static void main(String args[])
        {
                Parent p = new Parent();
                p.get(5,6);
                p.Show();
        }
}
```

Subclass constructor A subclass constructor is used to construct the instance variables of both the subclass and the superclass. The subclass constructor uses the keyword super to invoke the constructor method of the superclass. The keyword super is used as

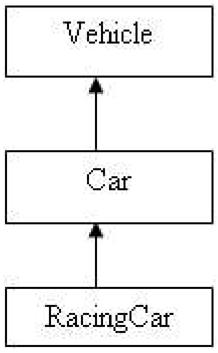
- Super may only be used within a subclass constructor method.
- The call to superclass constructor must appear as the first statement within the subclass constructor.
- The parameters in the super call must match the order and type of the instance variable declared in the superclass.

For Example, WAP that demonstrate the use of the keyword super

```java
class Room
{
        int length;
        int breadth;
        Room(int x, int y)
        {
                length= x;
                breadth = y;
        }
        int area()
        {
                return(length*breadth);
        }
}
class BedRoom extends Room
{
        int height;
        BedRoom(int
                        x, int y, int z)
        {
                super(x,y);
                //call the superclass constructor with value x and y
                height = z;
        }
        int volume()
        {
                return(length*breadth*height);
        }
}

class SuperTest
{
        public static void main(String args[])
        {
                BedRoom room1 = new BedRoom(2,3,4);//call subclass
                    constructor
                int area1 = room1.area();//superclass method
                int volume1 = room1.volume();//call subclass method
                System.out.println("Area of the room is= " +area1);
                System.out.println("Volume of the bedroom is = "
                    +volume1);
        }
}
```

Multilevel Inheritance When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance. The derived class is called the subclass or child class for it's parent class and this parent class works as the child class for it's just above (parent) class. Multilevel inheritance can go up to any number of levels.



Pictorial Representation of Simple and Multilevel Inheritance
In Multilevel Inheritance a derived class with multilevel base classes is declared as follows:

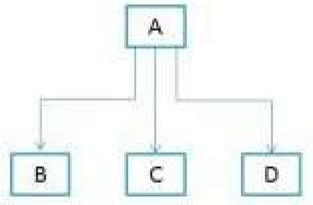
```
class vehicle{
        //Instance variables;
        //Methods;
}
class Car extends Vehicle{
        //first level
        //Instance variables;
        //Methods;
        //..............

}
class RacingCar extends Car{
        //second level
        //Instance variables;Methods;
        //.................................
}
```

Q. WAP that demonstrate the concept of multilevel inheritance class

```
class Student{
        int roll_number;
        Student(int x){
                roll_number = x;
        }
        void put_number(){
                System.out.println("Roll number = " +roll_number);
        }
}
class Test extends Student{
        double sub1;double sub2;
        Test(int a, double b, double c){
                super(a);
                sub1 = b;sub2 = c;
        }
        void put_marks(){
                System.out.println("Marks in subject first is = "
                    +sub1);
                System.out.println("Marks in subject second is = "
                    +sub2);
        }
}
class Result extends Test{
        double total;
        Result(int a, double b, double c){
                super(a,b,c);
        }void display(){
                total = sub1 +sub2;
                put_number();
                put_marks();
                System.out.println("Total marks = " +total);
        }
}
class MultiLevelDemo{
        public static void main(String args[]){
                Result ram = new Result(5,55.5,67.5);
                ram.display();
        }
}
```

Hierarchical Inheritance As you can see in the diagram below that when a class has more than one child classes (sub classes) or in other words more than one child classes have the same parent class then such kind of inheritance is known as hierarchical



Hierarchical Inheritance

Syntax:

```
class A{
       //Instance variables;
       //Methods;
}
class B extends A{
       //Instance variables of Parent and its own;
       //Methods of parent and its own
       //..............
}
class C extends A{
       //Instance variables of Parent and its own;
       //Methods of parent and its own
       //..............
}
class D extends A{
       //Instance variables of Parent and its own;
       //Methods of parent and its own
       //..............
}
```

WAP to demonstrate the hierarchical inheritance

```
class A{
       public void methodA(){
              System.out.println("method of Class A");
       }
}
class B extends A{
        public void methodB(){
                System.out.println("method of Class B");
        }
}
class C extends A{
        public void methodC(){
                System.out.println("method of Class C");
        }
}
class D extends A{
        public void methodD(){
                System.out.println("method of Class D");
        }
}
class MyClass{
        public void methodB(){
                System.out.println("method of Class B");
        }
        public static void main(String args[]) {
                B obj1 = new B();
                C obj2 = new C();
                D obj3 = new D();
                obj1.methodA();
                obj2.methodA();
                obj3.methodA();
        }
}
```

Java Does not support Multiple and Hybrid inheritance directly it is supported by package and interface

2.4.3 Visibility Control in Inheritance

There are mainly three types of visibility control that are used to restrict the access to certain variables and methods from outside the class. The visibility modifiers are also known as access modifiers. They provide different level of protection as described below they follow the following syntax.
Syntax and Example:

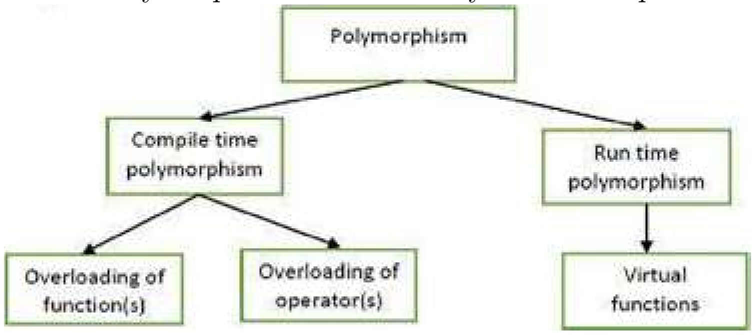
```
visibility-control type variablename;
visibility-control type methodname(arglist){
        //code block
}
```

```
Example,
private int rollNumber;
public int getRollNumber(){
       return rollNumber;
}
```

Public Access Members (Variables or methods) that are declared as public are known as public member of the class. Those members shave widest level of visibility. Those members that are declared as public can have access from same class, subclass in same package, other classes in same package, subclass in other packages and non-subclasses in other packages.
Protected Access Members that are declared as protected are known as protected members. The protected modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages. Protected members are not accessible from non-subclasses in other packages.
Private Access Members that are declared as private are known as private members. The private modifier makes field visible only within their own class. They cannot be inherited by subclasses and therefore not accessible in subclasses.

2.4.4 Polymorphism

Polymorphism is an important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. Polymorphism means, "One name, multiple forms". Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific action associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.


Polymorphism
Compile time polymorphism
Run time polymorphism
Overloading of function(s)
Overloading of operator(s)
Virtual functions

Compile time/Static polymorphism refers to the binding of functions on the basis of their signature (number, type and sequence of parameters). It is

also called early binding. In this the compiler selects the appropriate function during the compile time. The examples of compile time polymorphism are Function overloading (use the same function name to create functions that perform a variety of different tasks) and Operator overloading (assign multiple meanings to the operators).
Dynamic or Subtype or Runtime Polymorphism means the change of form by entity depending on the situation. If a member function is selected while the program is running, then it is called run time polymorphism. This feature makes the program more flexible as a function can be called, depending on the context. This is also called late binding. The example of run time polymorphism is virtual function.

2.5 Interface and Abstract class

2.5.1 Interface

An interface is a contract for what the classes can do. It, however, does not specify how the classes should do it. When a class implements a certain interface, it promises to provide implementation to all the abstract methods declared in the interface. Interface defines a set of common behaviors. The classes implement the interface agree to these behaviors and provide their own implementation to the behaviors. This allows you to program at the interface, instead of the actual implementation. One of the main usages of interface is provide a communication contract between two objects. If you know a class implements an interface, then you know that class contains concrete implementations of the methods declared in that interface, and you are guaranteed to be able to invoke these methods safely. In other words, two objects can communicate based on the contract defined in the interface, instead of their specific implementation.

2.5.2 Abstract Class

A class containing one or more abstract methods is called an abstract class. An abstract class is incomplete in its definition, since the implementation of its abstract methods is missing. Therefore, an abstract class can not be instantiated. In other words, you cannot create instances from an abstract class (otherwise, you will have an incomplete instance with missing method's body).
To use an abstract class, you have to derive a subclass from the abstract

class. In the derived subclass, you have to override the abstract methods and provide implementation to all the abstract methods. The subclass derived is now complete, and can be instantiated. (If a subclass does not provide implementation to all the abstract methods of the superclass, the subclass remains abstract.)