CHAPTER – 2

JAVA ARCHITECTURE AND OOPS

Java Architecture:

Compilation in Java:

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.

Java Virtual Machine (JVM):

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.

Why Java is Platform Independent?

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. We 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.

Java Runtime Environment (JRE) and Java Architecture in Detail:

Java Runtime Environment contains JVM, class libraries and other supporting components. As we know the Java source code is compiled into byte code by Java compiler. This byte code will be stored in class files. During runtime, this byte code will be loaded, verified and JVM interprets the byte code into machine code which will be executed in the machine in which the Java program runs.

A Java Runtime Environment performs the following main tasks respectively.

- **Loads The Class:** This is done by the class loader
- **Verifies The Byte Code:** This is done by byte code verifier.
- **Interprets The Byte Code:** This is done by the JVM

These tasks are described in detail in the subsequent sessions. A detailed Java architecture can be drawn as given below:

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 byte code is loaded successfully, then next step is byte code verification by byte code verifier.

Byte Code Verifier:

The byte code 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:

- The code follows JVM specifications.
- There is no unauthorized access to memory.
- The code does not cause any stack overflows.
- 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.

Just in Time Compiler:

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 byte code 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 byte code 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.

Why Java is Secure?

As we 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.

Garbage Collection:

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.

Java Classes and Objects:

Class:

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class is declared by use of the **class** keyword. The class body is enclosed between curly braces {}. The data or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Object:

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 an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, 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, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance (result) of a class.

Example Program for Class and Object:

```
class Student{
```

```
 int id;
  String name;
  public static void main(String args[]){
   Student s1=new Student();
   System.out.println(s1.id);
   System.out.println(s1.name);
  }
 }
```

Difference between Class and Object:

Object	Class
Object is an instance of a class.	Class is a blueprint or template from which objects are created.
Object is a real world entity such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a group of similar objects.
Object is a physical entity.	Class is a logical entity.
Object is created through new keyword mainly e.g. Student s1=new Student();	Class is declared using class keyword e.g. class Student{}
Object is created many times as per requirement.	Class is declared once.
Object allocates memory when it is created.	Class doesn't allocated memory when it is created.
There are many ways to create object in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only one way to define class in java using class keyword.

Class Methods and Instances:

Class Method:

A method is a collection of statements that perform some specific task and return result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class which is different from languages like C, C++ and Python. Methods are **time savers** and help us to **reuse** the code without retyping the code.

Method Declaration:

In general, method declarations has six components:

1. **Modifier**:

Defines **access type** of the method i.e. from where it can be accessed in our application. In Java, there are 4 type of the access specifiers.

a. **public:** accessible in all class in our application.
b. **protected**: accessible within the class in which it is defined and in its **subclass(es)**
c. **private:** accessible only within the class in which it is defined.
d. **default (declared/defined without using any modifier):** accessible within same class and package within which its class is defined.

2. **The Return Type :**

The data type of the value returned by the method or void if does not return a value.

3. **Method Name:**

The rules for field names apply to method names as well, but the convention is a little different.

4. **Parameter List:**

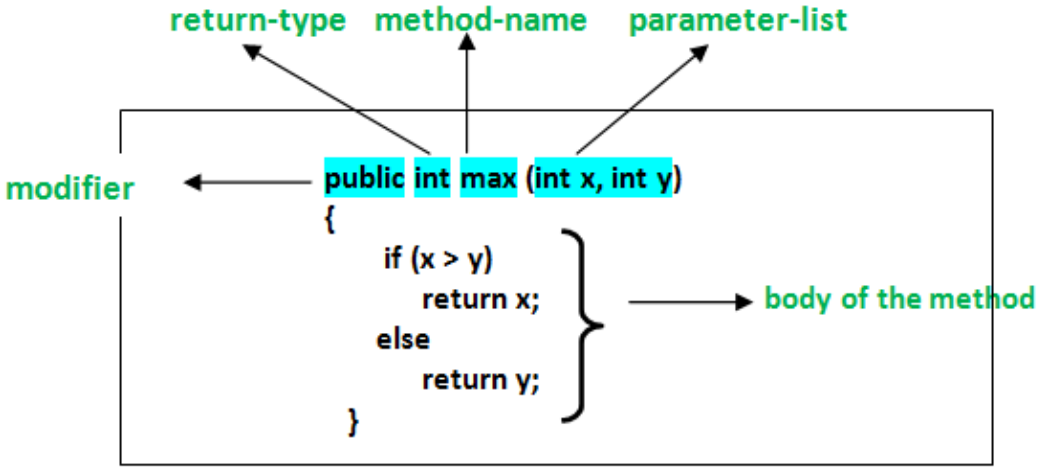
Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, we must use empty parentheses ().

5. **Exception List:**

The exceptions we expect by the method can throw, we can specify these exception(s).

6. **Method Body :**

It is enclosed between braces. The code we need to be executed to perform our intended operations.



Instances Methods:

Instance method are methods which require an object of its class to be created before it can be called. To invoke a instance method, we have to create an Object of the class in within which it defined.

```
public void geek(String name)
{
```

```
    // code to be executed....
}
// Return type can be int, float String or user defined data type.
```

Memory Allocation:

These methods themselves are stored in Permanent Generation space of heap but the parameters (arguments passed to them) and their local variables and the value to be returned are allocated in stack. They can be called within the same class in which they reside or from the different classes defined either in the same package or other packages depend on the **access type** provided to the desired instance method.

Important Points:

- Instance method(s) belong to the Object of the class not to the class i.e. they can be called after creating the Object of the class.
- Every individual Object created from the class has its own copy of the instance method(s) of that class.
- They can be overridden since they are resolved using **dynamic binding** at run time.

Example:

```
import java.io.*;
class Foo{
    String name = "";
    // Instance method to be called within the same class or
    // from a another class defined in the same package
    // or in different package.
    public void geek(String name){
        this.name = name;
    }
}

class GFG {
    public static void main (String[] args) {
        // create an instance of the class.
        Foo ob = new Foo();
        // calling an instance method in the class 'Foo'.
        ob.geek("GeeksforGeeks");
        System.out.println(ob.name);
    }
}
```

Inheritance and Polymorphism in Java:

Inheritance in Java:

Inheritance in Java is a mechanism in which one object acquires all the properties and behavior of a parent object. It is an important part of OOPs (Object Oriented programming

system). The idea behind inheritance in Java is that we can create new classes that are built upon existing classes. When we inherit from an existing class, we can reuse methods and fields of the parent class. Moreover, we can add new methods and fields in our current class also.

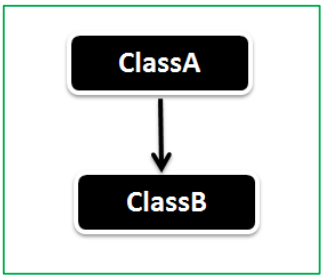
Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Syntax:

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

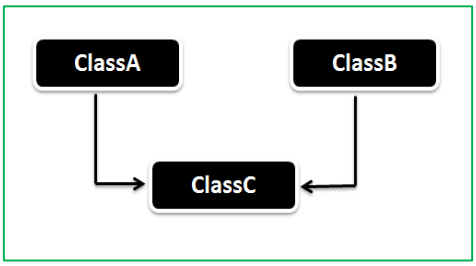
Types of Inheritance:

1. Single Inheritance in Java:



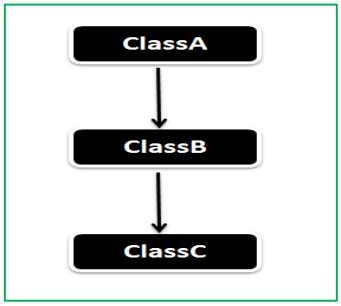
Single Inheritance is the simple inheritance of all, when a class extends another class (Only one class) then we call it as Single inheritance. The below diagram represents the single inheritance in java where Class B extends only one class Class A. Here Class B will be the Sub class and Class A will be one and only super class.

2. Multiple Inheritance in Java:



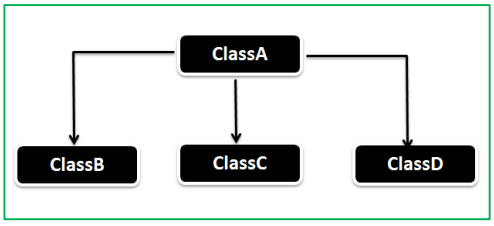
Multiple Inheritance is nothing but one class extending more than one class. Multiple Inheritance is basically not supported by many Object Oriented Programming languages such as Java, Small Talk, C# etc. (C++ Supports Multiple Inheritance). As the Child class has to manage the dependency of more than one Parent class. But we can achieve multiple inheritance in Java using Interfaces.

3. Multilevel Inheritance in Java:



In Multilevel Inheritance a derived class will be inheriting a parent class and as well as the derived class act as the parent class to other class. As seen in the below diagram. ClassB inherits the property of ClassA and again ClassB act as a parent for ClassC. In Short ClassA parent for ClassB and ClassB parent for ClassC.

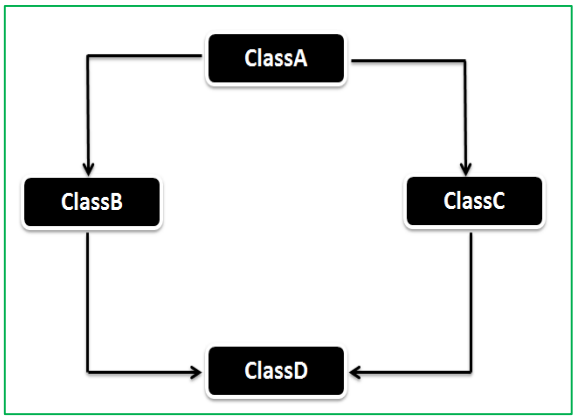
4. Hierarchical Inheritance in Java:



In Hierarchical inheritance one parent class will be inherited by many sub classes. As per the below example ClassA will be inherited by ClassB, ClassC and ClassD. ClassA will be acting as a parent class for ClassB, ClassC and ClassD

5. **Hybrid Inheritance in Java:**

Hybrid Inheritance is the combination of both Single and Multiple Inheritance. Again Hybrid inheritance is also not directly supported in Java only through interface we can achieve this. Flow diagram of the Hybrid inheritance will look like below. As you can ClassA will be acting as the Parent class for ClassB & ClassC and ClassB & ClassC will be acting as Parent for ClassD.



Polymorphism in Java:

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding. If we overload a static method in Java, it is the example of compile time polymorphism.

Types of Polymorphism:

1. **Compile Time/ Static Polymorphism:**

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).

2. **Dynamic or Subtype or Runtime Polymorphism:**

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.

Method Overloading:

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**. If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose we have to perform addition of the given numbers but there can be any number of arguments, if we write the method such as a(int, int) for two parameters, and b(int, int, int) for three parameters then it may be difficult for you as well as other programmers to

understand the behavior of the method because its name differs. So, we perform method overloading to figure out the program quickly.

Example:

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Method Overriding:

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**. In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Example:

```
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Interface and Abstract Class:

Interface in Java:

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

In other words, we can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**. It cannot be instantiated just like the abstract class. Since Java 8, we can have **default and static methods** in an interface. Since Java 9, we can have **private methods** in an interface.

Why Use Java Interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Syntax:

```
interface <interface_name>{
   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

Example:

```
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();
 }
}
```

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember:

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example:

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Difference:

Abstract class	Interface
Abstract class can have abstract and non-abstractmethods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
An abstract classcan extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
An abstract classcan be extended using keyword? extends?	An interface classcan be implemented using keyword? implements?
A Javaabstract classcan have class members like private, protected, etc.	Members of a Java interface are public by default.

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

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).