1. **Overview of JVM Architecture**

Refer to diagrams : JVM Architecture Overview.png, JVM Architecture in brief.jpg

Main job of JVM is

- Load and execute Java Program.

JVM has various sub components internally.

1.1 **Class loader sub system**: JVM's class loader sub system performs 3 tasks

a. It loads .class file into memory.

b. It verifies byte code instructions.

c. It allots memory required for the program.

1.2 **Run time data area**: When Java application starts running, underlying O.S (operating system) allocates some memory. Exact details are dependent upon specific JDK version.

This is the memory resource used by JVM and it is divided into 5 parts

a. Method area: **Class-level metadata storage** (class definitions, method definitions, runtime constant pools, field/method metadata).

   Earlier it was implemented as PermGen Space . (Now it's implemented as meta space since Java SE 8)

b. Heap: Objects are created on heap.

c. Java stacks: Java stacks are the places where the Java methods are executed. A Java stack contains frames. On each frame, a separate method is executed.

d. Program counter registers: The program counter registers store memory address of the instruction to be executed by the micro processor.

e. Native method stacks: The native method stacks are places where native methods (for example, C language programs) are executed. Native method is a function, which is written in another language other than Java.

Typically system with 16 GB RAM, default heap would start at ~256 MB(1/64 of physical memory) and could grow up to ~4 GB(1/4 of physical memory) if not overridden. Meta space still has no preset max , it grows until system memory is exhausted. Default per-thread stack size = 1 MB on most 64-bit platforms.Code

Cache has initial size = 2 MB , Reserved = 240 MB (on 64-bit HotSpot).

1.3 **Native method interface**: Native method interface is a program that connects native methods libraries (C header files) with JVM for executing native methods.

1.4 **Native method library**: holds the native libraries information.

1.5 **Execution engine**: Execution engine contains interpreter and JIT (Just In Time) compiler, which converts byte code into machine code. JVM uses optimization technique to decide which part to be interpreted and which part to be used with JIT compiler. The HotSpot represents the block of code executed by JIT compiler.

2. OOP (Object-Oriented Programming)

It is a style of programming that breaks the program on the basis of the objects in it.

It mainly works on Class, Object, Polymorphism, Abstraction, Encapsulation and Inheritance.

Its aim is to bind(encapsulate) together the data and functions(methods) that operate on them.

Some of the well-known object-oriented languages are Perl, Java, Python, C++ , C# etc

Benefits of OOP

1. One can build the programs from standard working modules that communicate with one another, instead of writing the code from scratch .It saves lot of development time and offers higher productivity.

2. OOP language allows to break the program into smaller-sized problems that can be solved easily (one object at a time).

3. Upgrading, extending & testing the application is easier.

4. It is very easy to partition the work in a project based on objects.

5. Data hiding principal helps the programmer to build secure programs which cannot be modified by the code in other parts of the program.

6. By using inheritance, one can eliminate redundant code and reuse existing classes.

7. Using abstraction , complex implementations can be hidden from the users.

Class & Object

Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.

Class

A class is a user defined blueprint or prototype from which objects are created.

It represents the set of properties (state) and methods(behavior) that are common to all objects of one type.

Class declaration includes

1. Access specifiers : A class can have public or default access .

2. Class name: The name should begin with a capital letter & then follow camel case convention.

3. Superclass(if any): The name of the class parent (superclass), if any, preceded by the keyword extends. (Implicit super class of all java classes is java.lang.Object)

4. Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.

eg : public class Emp extends Person implements Sportsman,Artist{...}

5. Body: The class body surrounded by braces, { }.

6. Constructors are used for initializing state of new objects.

7. Fields are variables that provides the state of the class and its objects.

8. Methods are used to implement the behavior of the class and its objects.

eg : Student,Employee,Flight,PurchaseOrder, Shape ,BankAccount

Object

It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects, which interact by invoking methods.

An object consists of :

State : It is represented by attributes of an object. (Properties of an object) / instance variables(non static data members)

Behavior: It is represented by methods of an object (actions upon data)

Identity: It gives a unique identity to an object and enables one object to interact with other objects. eg : Employee id / Student PRN / Invoice No

Creating an object

The **new** (keyword) operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

Constructor is a special method having

- same name as the class name
- no explicit return type
- may be parameterized or parameter less.

Parameterized constructor is used initialize (init) state of the object.

If a class does not explicitly declare any constructor , the Java compiler automatically provides a no-argument constructor, called the default constructor.

This default constructor implicitly calls the super class's no-argument constructor and initializes the complete state to default values.

What is "**this**"?

- A keyword
- It represents , a current object reference.

Usages of this

1. To unhide, instance variables from method local variables.(to resolve the name conflict)

Eg. : In a typical Person class , to initialize a name

- this.name=name;
- LHS(Left hand side) refers to instance variable and RHS(Right Hand side) refers to a local variable name.

2. To invoke the constructor, from another overloaded constructor in the same class.(constructor chaining , to avoid code duplication)

Encapsulation in Java

Encapsulation is defined as the wrapping up of data & code under a single unit. It is the mechanism that binds together code and the data it manipulates.

It's is a protective shield that prevents the data from being accessed by the code outside this shield.

The variables or data of a class is hidden from any other class and can be accessed only through any member function/method of own class in which they are declared.

As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.

Tight Encapsulation can be achieved by

- Declaring all the variables in the class as private and writing public methods as their accessors.

Advantages of Encapsulation:

1. Data Hiding (security)

2. Increased Flexibility: We can make the variables of the class as read-only or write only or readable and writable

3. Reusability: Encapsulation also improves the re-usability and easy to change with new requirements.

4. Testing code is easy.

Information hiding is achieved by private data members & supplying public accessors.

How Encapsulation Achieves Abstraction ?

Eg.

```
class BankAccount {

    private String accountNumber;    // hidden from outside the class

    private double balance;          // hidden from outside the class

    // Constructor

    public BankAccount(String accountNumber, double initialBalance) {

        this.accountNumber = accountNumber;

        this.balance = initialBalance;

    }
```

```
    // Public methods - interface to the outside world (Abstraction)
    public void deposit(double amount) {
        //Business logic implementation
    }
    public void withdraw(double amount) {
       //Business logic implementation
    }
    public double getBalance() {
        return balance;
    }
}
```

1. **Hides Internal Details**

- By marking fields private and providing controlled access via public methods, the **implementation details are hidden**.
- Eg: In BankAccount class , balance cannot be modified directly → prevents inconsistent states.

2. **Provides a Simple Interface**

- Public methods act as a **contract**: users can only interact through these methods.
- Users don't need to know **how the deposit calculation works internally**.

3. **Separates "What" from "How"**

- Abstraction = "What does the object do?"
- Encapsulation = "How do I hide the details while letting users do it safely?"
- Together, they allow **focus on functionality without exposing complexity**.

3. **Garbage Collection** :

Garbage Collection(GC) is a process to identify and delete the objects from Heap memory which are not in use. GC frees the space after removing unreferenced objects.

Typically an unreachable object (valid no. of references reduced to 0) is termed as garbage.

Java Programmers **don't have to manually free memory** (like free() in C/C++).

- **prevents memory leaks and optimizes memory usage**.

Memory Leak in Java

- Occurs when **objects are no longer needed but** JVM cannot reclaim their memory.

When JVM starts:

1. It spawns the **main thread** (user thread) to run main method.
2. It spawns **GC daemon threads** to manage memory automatically.
3. It spawns a **finalizer daemon thread** (special daemon thread) to execute finalize() methods for objects that override it.

Even if no object overrides finalize(), the **finalizer thread exists**, but it stays idle until needed.

GC threads will be automatically given a chance, when heap is under pressure(i.e less heap memory is available)

How to request for GC explicitly (not recommended)?

API of System class

public static void **gc()**

eg : System.gc();

Object class API

protected void **finalize()** throws Throwable

This method is automatically called by the GC threads on an object before destroying the object .

It is **deprecated** because

- It's not guaranteed to run
- Can cause performance overhead for JVM GC threads

Releasing of non-Java resources

- Eg. closing of DB connection, closing file handles,closing socket connections etc is NOT done automatically by GC threads.
- You should not do it in finalize method.

Triggers for making the object eligible for GC(candidate for GC)

1. Nullifying all valid references to any object

eg : Box b1=new Box(1,2,3);

Box b2=b1;

Box b3=b1;

b1=b2=b3=null; // 1st Box object marked for GC

2. Re-assigning the reference to another object

eg : Box b1=new Box(10,20,30);

b1=new Box(11,21,31);//1st Box marked for GC

3. Object created within a method & its reference is NOT returned to the caller.

4. Island of isolation