1. **What does JRE contains?**

Ans:-

JRE contains JVM and API.

JVM is a software responsible for executing any kind of Java application.

API consists of jar files and dll files.

Jar files consists of .class files for inbuilt classes java has got. Dll files consists of native code ( c,c++ code ) which java uses internally.

1. **What are the 3 main components of JVM Architecture?**

Ans:

1. Class loader subsystem
2. Runtime memory areas
3. Execution engine
4. **Explain class loader subsystem.**

Ans:

**Loading**

* + Loading is the process of finding the binary representation of a class or interface type with a particular name.
  + For each .class file JVM stores information in the **Method area or Metaspace**.
  + After loading .class file JVM creates an object of type **java.lang.Class** to represent the loaded class in the heap memory.

[ JVM follows **delegation-hierarchy** principle to load the classes]

**Linking**

* Verify

BytecodeVerifier is going to check whether ur generated bytecode is proper or not. Ie. Whether it contains any virus or malicious code and whether class file format is compatible with JVM class specification. If verification fails then we will get “Verify Error”. Java program is always secure. You can execute on any machine happily because Bytecode Verifier is going ensure whether bytecode is valid or not.

* Prepare

For static variables memory will be allocated in the **Method area** or **Metaspace** and assigned with the default values. (not the values you have initialized with)

e.g.

public class Vehicle

{

public static int wheels=2;

}

What preparation phase will do?

Method Area or Metaspace

Vehicle

Metadata

**wheels=0**

* Resolve

All symbolic references are replaced with original references from “Method Area”. Suppose u have references to other classes, these are changed from symbolic to the actual reference.

**Initialization**

Static members are given the values with which programmer has initialized them. Static initializers are executed. Both static members initialization and static block or initializer invocation happens in the order in which they are defined.

1. **What are various Runtime memory areas in JVM?**

Ans:

**Runtime Memory Areas in JVM**

The **Java Virtual Machine (JVM)** divides memory into different runtime areas to manage the execution of Java programs efficiently. The main memory areas are:

**1. Method Area (Metaspace in Java 8+)**

**Stores:**

* Class metadata (including fully qualified class names, method and field names, and bytecode).
* Runtime constant pool (including literals and references).
* Static variables.

**2. Heap Area**

**Stores:**

* **All objects and instance variables.**
* **It’s Shared** among all threads.
* Divided into **Young Generation (Eden + Survivor Spaces) and Old Generation (Tenured Space).**
* **Garbage Collection** manages unused objects.

**3. Stack Area**

**Stores:**

* **Method call frames** (local variables, parameters, return addresses).
* **Each thread gets its own stack** (not shared).
* **Causes StackOverflowError if it exceeds allocated memory.**

**4. PC Register (Program Counter Register)**

**Stores:**

* The **address of the currently executing JVM instruction**.
* **Each thread has its own PC register.**
* Helps in **switching between threads**.

**5. Native Method Stack (C Stack)**

**Stores:**

* Native method calls (methods written in C/C++ using JNI - Java Native Interface).
* Each thread has a separate **native stack**.
* Used for interfacing Java with **native libraries** (e.g., C/C++).

1. **Explain Execution engine inside JVM.**

Ans:

**Execution Engine in JVM**

The **Execution Engine** is the **core component of the JVM** responsible for executing Java bytecode. It takes the bytecode (from .class files) and **converts it into machine code** that the CPU can understand.

**Components of the Execution Engine**

The **Execution Engine** consists of the following key components:

**1. Interpreter**

**What it does?**

* **Reads and executes bytecode line-by-line**.
* **Slower** because it **translates bytecode every time it's executed**.
* Used in the **initial phase** of execution.

**Example**

for (int i = 0; i < 1000; i++) {

System.out.println(i);

}

* The **interpreter executes each loop iteration separately**, translating the same bytecode multiple times, which makes it **slow**.

**2. HotSpot Profiler**

**What it does?**

* **Monitors** program execution to find **frequently used code** (hot spots).
* Determines which methods should be **optimized** by the JIT compiler.

**Example**

* If a method is executed **thousands of times**, HotSpot marks it as **"hot"**, and the JIT compiler **compiles it into native code**.

**3. Just-In-Time (JIT) Compiler**

**What it does?**

* **Converts hot methods into native machine code** for faster execution.
* **Avoids re-interpreting bytecode** for frequently used methods.
* **Uses optimizations** like inlining, loop unrolling, and dead code elimination.

**How it Works (Step-by-Step)**

1. Initially, JVM **interprets all code**.
2. **HotSpot Profiler detects hot methods**.
3. **JIT Compiler compiles** these methods into **native machine code**.
4. The **JVM reuses the compiled code**, making execution faster.

**4. Garbage Collector (GC)**

**What it does?**

* **Automatically removes unused objects** to free memory.
* Runs in the **background** to optimize memory usage.
* Prevents **memory leaks**.

**Example**

String name = new String("Java");

// If `name` is no longer used, GC removes it from memory.

**How the Execution Engine Works (Complete Flow)**

**Step-by-Step Execution Process**

1️ **Class Loader loads .class files** into JVM.  
2️ **Execution Engine starts interpreting** the bytecode line-by-line.  
3️ **HotSpot Profiler analyses execution** and detects frequently used (hot) methods.  
4️ **JIT Compiler compiles hot methods** into native code for better performance.  
5️ **Garbage Collector cleans up** unused objects from memory.

1. **What is the difference between javac compiler and JIT compiler?**

Ans:

Javac compiler converts source code into bytecode whereas JIT compiler converts bytecode into native code.

1. **What is the difference between interpreter and JIT compiler?**

Ans:

**Difference Between Interpreter and JIT Compiler**

Both **Interpreter** and **JIT (Just-In-Time) Compiler** are part of the **JVM Execution Engine**, and their job is to **convert Java bytecode into machine code** so that the CPU can execute it. However, they work differently.

**1. Interpreter (Slow but starts execution quickly)**

**How it works?**

* Reads and executes **bytecode line by line**.
* **Translates the same code repeatedly** every time it is executed.
* Slower because it doesn’t save compiled machine code.

**Example:**  
Think of an interpreter as a **real-time translator** at a conference. Every time someone speaks, the translator **translates word by word** for the audience. This takes time.

**2. JIT (Just-In-Time) Compiler (Fast but takes time to optimize)**

**How it works?**

* Analyses code and **compiles frequently used methods into native machine code**.
* The compiled machine code is **saved and reused**, making execution much faster.
* Helps improve performance over time.

**Example:**  
Think of the JIT compiler as **preparing subtitles** for a movie. Instead of translating in real-time, subtitles are prepared **once** and can be **reused** every time someone watches the movie.