### 1. What is JVM?

Define JVM and explain its role in Java application execution

The Java Virtual Machine (JVM) is an abstract computing machine that provides a runtime environment for executing Java bytecode. It acts as a bridge between Java applications and the underlying hardware and operating system, allowing Java code to be executed across various platforms without modification.

Role in Java Application Execution:

1. **1. Bytecode Execution:**

The JVM's primary function is to load and execute Java bytecode (compiled Java code).

1. **2. Platform Independence:**

JVM enables "Write Once, Run Anywhere" (WORA) by interpreting bytecode on different platforms, ensuring consistent execution.

1. **3. Memory Management:**

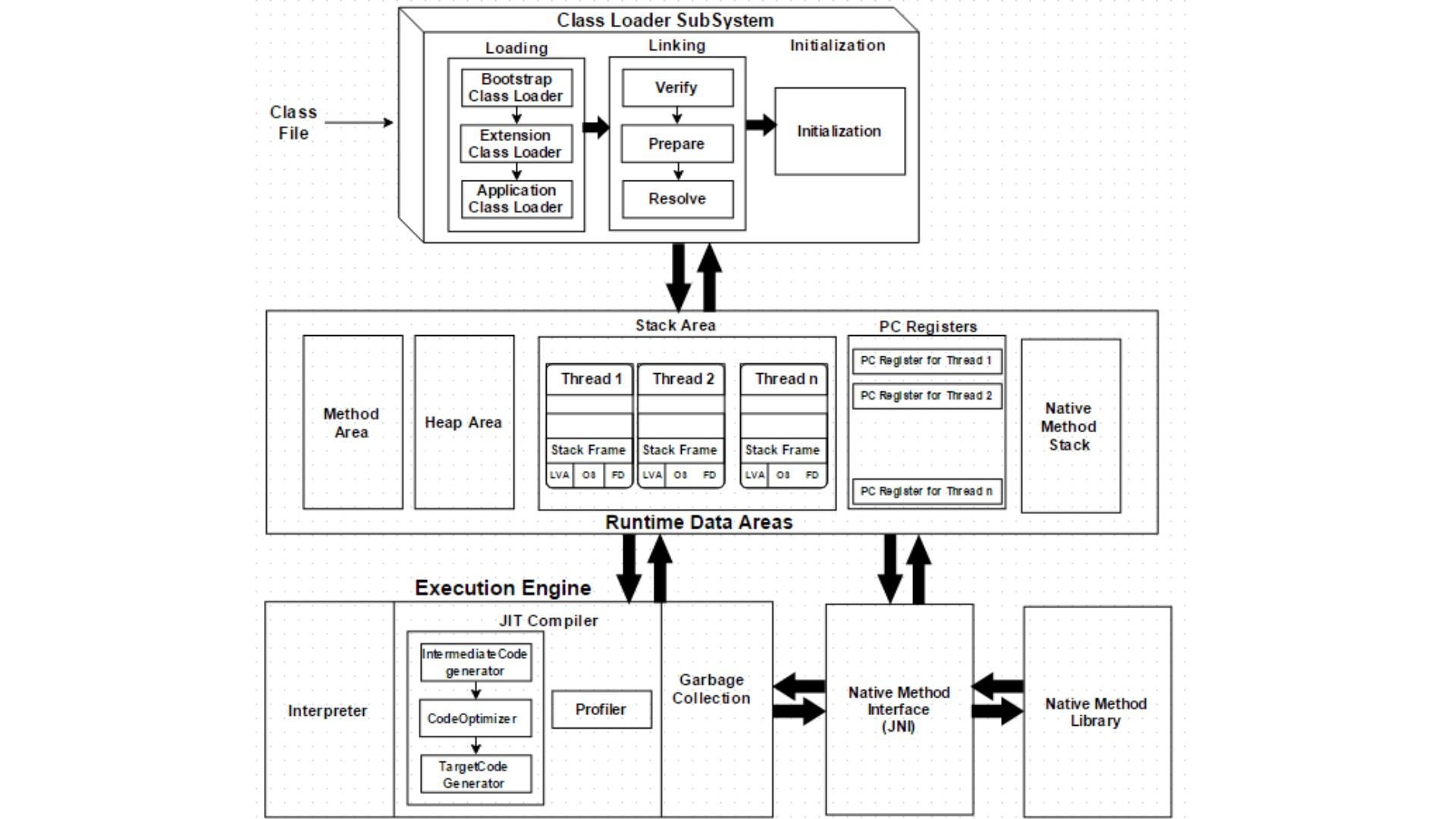
The JVM manages memory allocation and garbage collection, freeing developers from manual memory management.

1. **4. Security:**

JVM provides security features, such as security managers and access control, to protect the system from malicious code.

1. **5. Class Loading:**

The JVM loads and links Java classes into memory during execution, allowing dynamic loading of new classes at runtime.



### 2. Explain the Class Loader Subsystem with a Diagram

* Describe how the Class Loader loads classes.
* Include the three main components: Bootstrap, Extension, and Application Class Loaders.
* Provide a labeled diagram.

### ****Class Loader Subsystem in Java****

The **Class Loader Subsystem** in Java is responsible for loading, linking, and initializing classes and interfaces. When a Java program is executed, the class loader loads the .class files into memory for execution.

### ****Phases of Class Loading:****

1. **Loading**: Loads the class bytecode from the file system or network.
2. **Linking**:
   * **Verification**: Ensures bytecode is valid.
   * **Preparation**: Allocates memory for static variables and initializes them with default values.
   * **Resolution**: Replaces symbolic references with direct references.
3. **Initialization**: Initializes static variables with the actual values and executes static blocks.

### ****Three Main Class Loaders:****

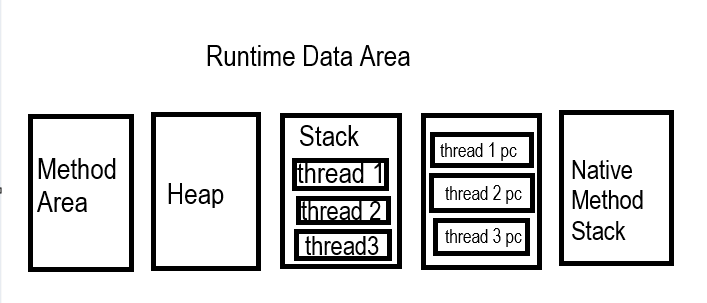
1. **Bootstrap Class Loader**:
   * **Loads**: Core Java classes (java.lang, java.util, etc.).
   * **Location**: jre/lib/rt.jar
   * **Implemented in**: Native code (part of JVM).
2. **Extension (Platform) Class Loader**:
   * **Loads**: Classes from the ext directory or extension directories (jre/lib/ext).
   * **Delegated by**: Bootstrap loader.
3. **Application (System) Class Loader**:
   * **Loads**: Classes from the application classpath (CLASSPATH variable).
   * **Delegated by**: Extension loader.

### ****Class Loader Hierarchy and Delegation Model:****

Each class loader delegates the class loading request to its parent before attempting to load the class itself. This ensures core classes are loaded by the Bootstrap loader only once and prevents reloading by user-defined loaders.

### 3. Explain Runtime Data Areas with a Diagram

* List and describe each of the runtime memory areas: Method Area, Heap, Stack, Program Counter Register, and Native Method Stack.
* Include a clear and labeled diagram.



### 🔷 Java Runtime Data Areas (Memory Areas in JVM)

When a Java program runs, the Java Virtual Machine (JVM) divides the memory into different sections known as Runtime Data Areas. These areas are used to manage data and control program execution.

### 📌 1. ****Method Area****

* **Purpose**: Stores class-level information (class metadata).
* **Contains**:
  + Class names
  + Method and field names
  + Bytecode of methods
  + Runtime constant pool
* **Shared among all threads**.

### 📌 2. ****Heap****

* **Purpose**: Stores **objects** and **class instances**.
* **Contains**:
  + All Java objects
  + Arrays
* **Shared among all threads**.
* This area is managed by the **Garbage Collector**.

### 📌 3. ****Java Stack (Thread Stack)****

* **Purpose**: Stores method **call frames**.
* **Contains**:
  + Local variables
  + Operand stacks
  + Return values
* **Each thread has its own stack**.
* Created at the same time as the thread.

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

* **Purpose**: Holds the **address of the current instruction** being executed.
* **Each thread has its own PC register**.
* Points to the bytecode instruction in the method.

### 📌 5. ****Native Method Stack****

* **Purpose**: Supports **native (non-Java) methods**, such as C/C++ methods called from Java.
* **Used for**:
  + Execution of platform-dependent code
* Each thread has its own native method stack.

### 4. Explain the Execution Engine with a Diagram

* Discuss the components of the execution engine: Interpreter, JIT Compiler, and Garbage Collector.
* Support your explanation with a detailed diagram.

### ****Execution Engine in Java****

The **Execution Engine** is a core component of the Java Virtual Machine (JVM) responsible for executing the bytecode of Java programs. Once the Class Loader loads classes into the JVM memory and they are verified and prepared, the Execution Engine takes over to run them.

### ****Main Components of the Execution Engine:****

#### 1. ****Interpreter****

* Reads and executes bytecode instructions one by one.
* **Pros**: Fast to start; useful for small or short-lived applications.
* **Cons**: Slower performance due to repeated interpretation of frequently executed code.

#### 2. ****JIT Compiler (Just-In-Time Compiler)****

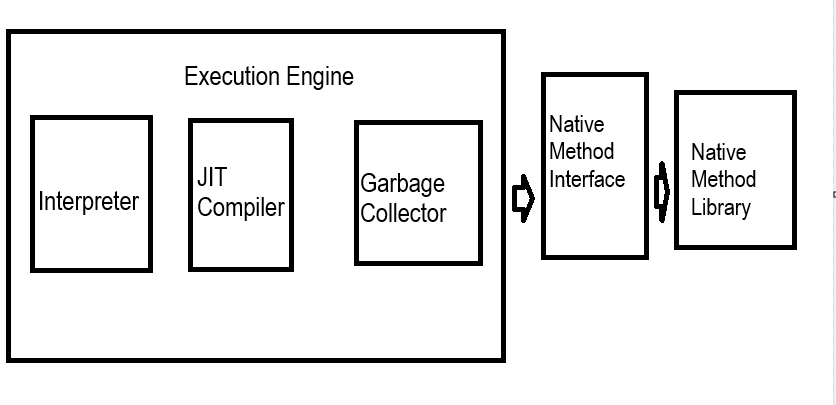
* Translates frequently executed bytecode into **native machine code** at runtime.
* Stores the compiled native code to avoid re-translation.
* **Pros**: Improves performance for long-running applications.
* **Working**: JVM monitors which code blocks are used often (called **hot spots**) and compiles them to native code.

#### 3. ****Garbage Collector****

* Manages memory by **automatically deallocating objects** that are no longer in use.
* Prevents **memory leaks** and ensures optimal memory utilization.
* Different types of garbage collectors exist (e.g., Serial, Parallel, G1, ZGC).

### ****Execution Flow Overview:****

1. **Class Loader** loads .class files.
2. Bytecode is **verified and prepared**.
3. **Execution Engine** executes the bytecode:
   * The **Interpreter** initially executes instructions.
   * The **JIT Compiler** optimizes frequently used code.
   * The **Garbage Collector** runs in the background to manage memory.



### 5. Java Code Example with Static Methods and Local Variables (with Diagram)

* Provide a simple Java code snippet using a static method and local variables.
* Include a diagram showing memory allocation for static context and local variables.



When the program starts, the entire class is first loaded into the Method Area, which stores the bytecode of all variables, methods, blocks, and constructors. Then, the Stack Area creates a stack frame for the main() method. Inside main(), the m1() method is invoked, which leads to the creation of a second stack frame specifically for m1(). Within m1(), a local variable is initialized and its value is printed. Since there are no further statements in m1(), its stack frame is removed after execution. Control then returns to the main() method. As there are no more statements to execute in main(), its stack frame is also destroyed. At this point, the Execution Engine completes the program’s execution.

### 6. Java Code Example with Non-Static Methods and Local Variables (with Diagram)

* Provide a simple Java code snippet using a non-static method and local variables.
* Include a diagram showing memory allocation for objects and method calls in the stack.



When the program begins, the entire class template is loaded into the Method Area, which holds the bytecode for all variables, methods, blocks, and constructors. In the Stack Area, a stack frame is created for the main() method. Inside main(), an object is created, which causes memory to be allocated in the Heap Area for all non-static members, and any non-static methods are loaded into the heap. A reference to this object is then created and stored in a reference variable. The main() method then calls m1(), leading to the creation of a new stack frame for m1(). The m1() method prints a value, and since it has no further statements, its frame is destroyed. Control returns to the main() method, which prints a statement, and once all remaining instructions are executed, its stack frame is also destroyed. This marks the end of the program execution.