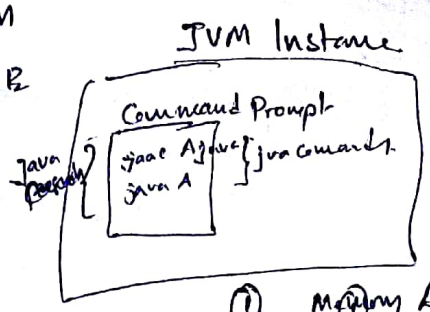
String pool

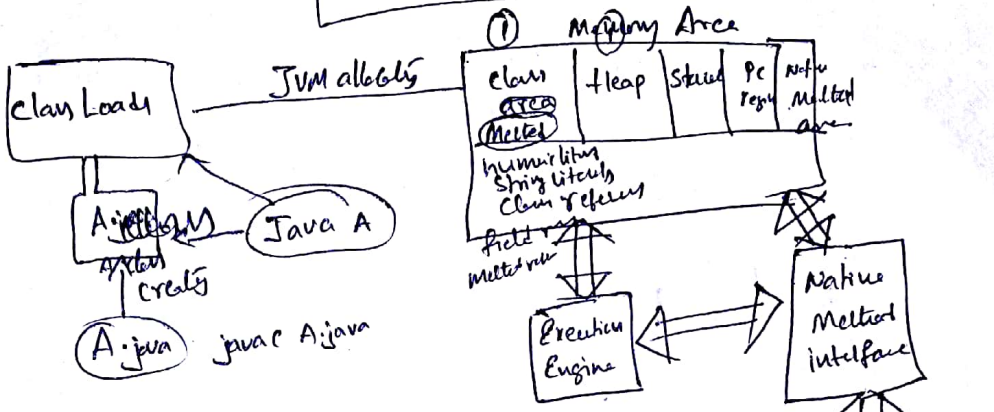
Specification → JVM

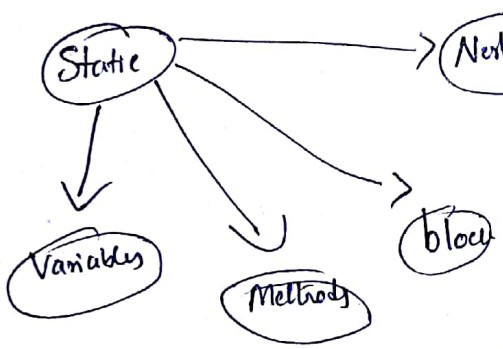
Implementation → JRE

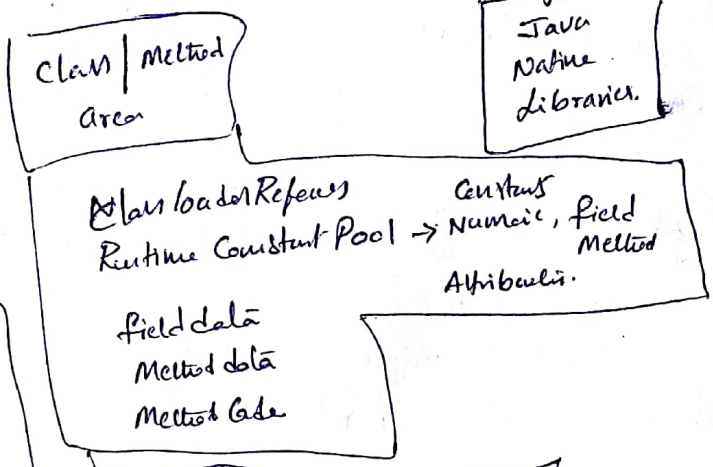
Runtime Instance. →

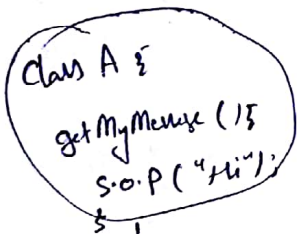


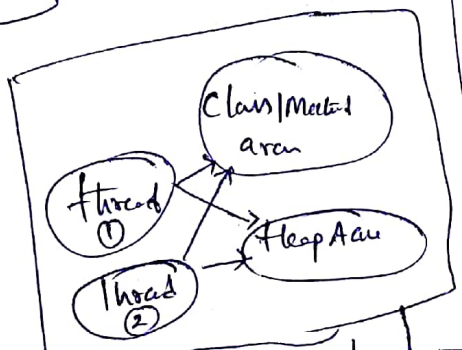

```
Class A {
    Static myName = "Hi";
    Static {
    int a;
    String getMessage()
    {
    return S.O.P("Hello");
    }
```

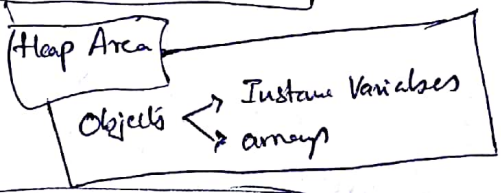


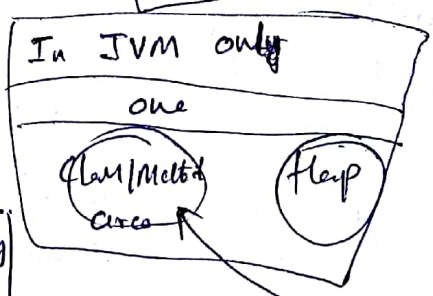



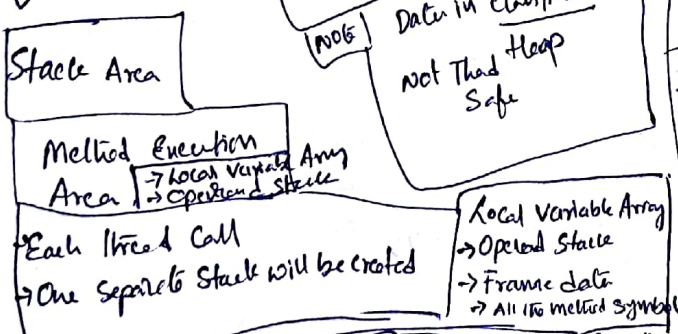



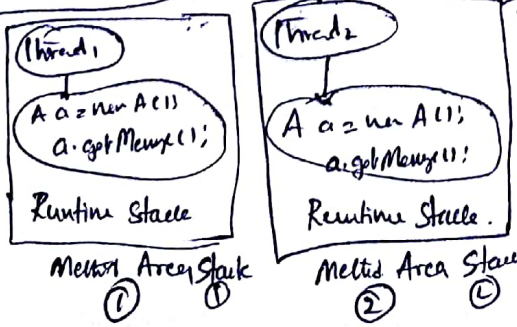



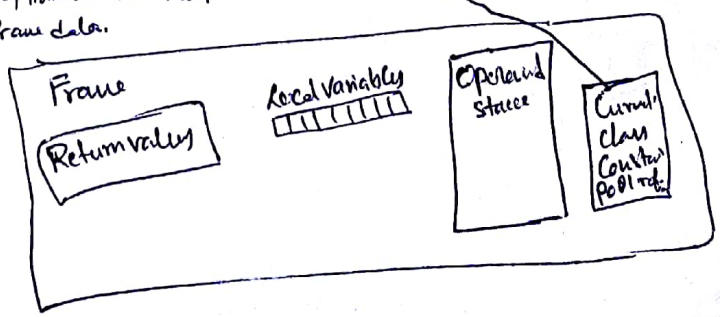



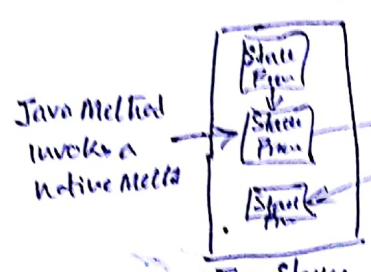

Scanned by CamScanner

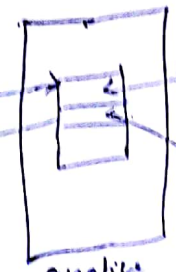
PC - Registers - Each thread will have separate PC Registers, to hold the address of Current Execution Instruction.
(Program Counter) once the instruction is executed the PC Register will be updated with the next instruction.

PC - Register → In a Computer processor that controls the Address (location) of the instruction being executed at the current time.

A Register → is one of a small set of data holding places that the processor uses.

Native Method Stacks: → Native Method Stacks holds native method information.
→ For every thread, a separate native method stack will be created.

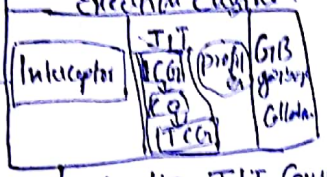



**Java Native Methods**

Native Methods are functions written in C or C++, compiled into a library, and dynamically loaded by the Java runtime.

## Execution Engine




* Interpreter → Interprets the bytecode faster, but executes slowly.
→ disv→ one method is called multiple times, every time a new interpretation is required.

JIT Compiler

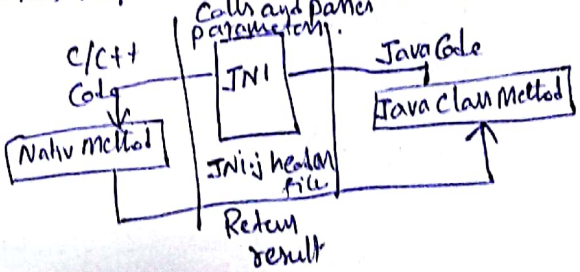
→ The JIT Compiler neutralizes the disadvantages of the Interpreter.
→ The Ex-Engine takes help of the Interpreter in converting byte Code.
→ but when it finds repeated Code it uses the JIT Compiler,
→ Which Compiles the entire bytecode and changes it to native Code.
→ the code directly for repeated method calls, which improves performance of the System.

bytecode into machine code. (Instructions)
→ ICG → Intermediate Code generator
→ CO → Code Optimizer
→ TCG → Target Code Generator
→ profiler

ICG → produces intermediate Code
CO → Responsible for optimizing the intermediate Code generated above
TCG → Responsible for Generating Machine Code or Native Code
profiler → A special Component, responsible for finding hotspots i.e. whether the method is called multiple times or not.

GC → Garbage Collector:
→ Collects and removes unreferenced Objects
→ GC can be triggered by calling "System.gc()" but the execution is not guaranteed.
→ GC of JVM collects the objects that are created.

JNI → Java Native Interface.
→ Interacting with the Native Method Libraries and provides the Native Libraries required for the Exe-Engine.




Native - Method Libraries

It is a collection of the Native Libraries which is required for the Execution Engine.

Scanned by CamScanner

# The JVM Architecture Explained

An overview of the different components of the JVM, along with a very useful diagram

by Jackson Joseraj · Sep. 02, 16 · Java Zone

Like (220) Comment (22) Save Tweet 161.8k Views

Learn how to troubleshoot and diagnose some of the most common performance issues in Java today. Brought to you in partnership with AppDynamics.

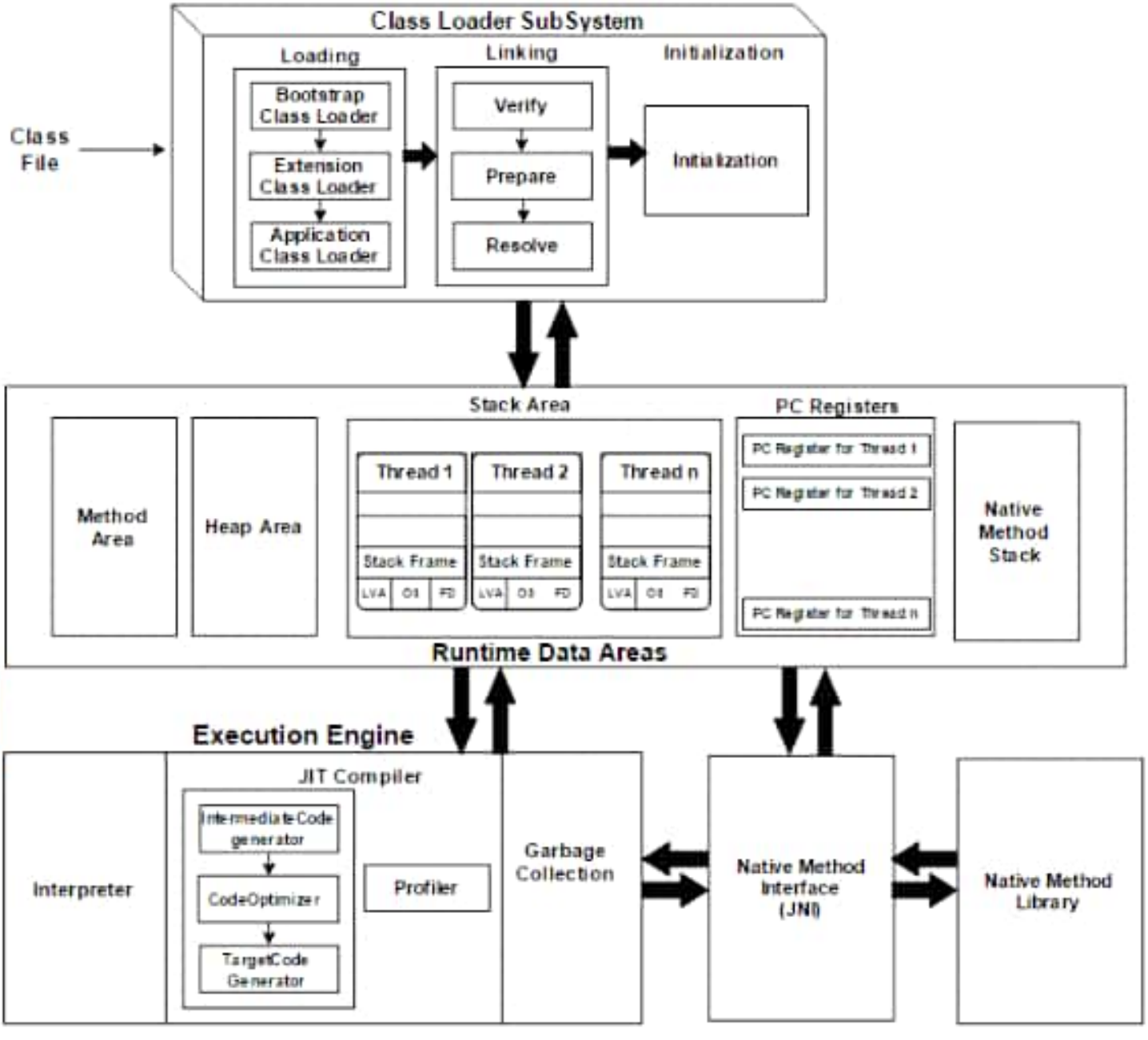
Every Java developer knows that bytecode will be executed by **JRE** (Java Runtime Environment). But many doesn't know the fact that **JRE** is the implementation of **Java Virtual Machine** (JVM), which analyzes the bytecode, interprets the code, and executes it. It is very important as a developer that we should know the Architecture of the JVM, as it enables us to write code more efficiently. In this article, we will learn more deeply about the **JVM architecture** in Java and the **different components** of the JVM.

## What is the JVM?

**A Virtual Machine** is a software implementation of a physical machine. Java was developed with the concept of **WORA (*Write Once Run Anywhere*)**, which runs on a **VM**. The **compiler** compiles the Java file into a Java **.class** file, then that **.class** file is input into the JVM, which Loads and executes the class file. Below is a diagram of the Architecture of the JVM.

Scanned by CamScanner

# JVM Architecture Diagram




Scanned by CamScanner

## How Does the JVM Work?

As shown in the above architecture diagram, the JVM is divided into three main subsystems:

1. **Class Loader Subsystem**
2. **Runtime Data Area**
3. **Execution Engine**

## 1. Class Loader Subsystem

Java's **dynamic class loading** functionality is handled by the class loader subsystem. It loads, links. and initializes the class file when it refers to a class for the first time at **runtime**, not **compile time.**

#### 1.1 Loading

Classes will be loaded by this component. Boot Strap class Loader, Extension class Loader, and Application class Loader are the three class loader which will help in achieving it.

1. **Boot Strap ClassLoader** – Responsible for loading classes from the bootstrap classpath, nothing but **rt.jar.** Highest priority will be given to this loader.
2. **Extension ClassLoader** – Responsible for loading classes which are inside **ext** folder **(jre\lib).**
3. **Application ClassLoader** –Responsible for loading **Application Level Classpath**, path mentioned Environment Variable etc.

The above **Class Loaders** will follow **Delegation Hierarchy Algorithm** while loading the class files.

#### 1.2 Linking

1. **Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the **verification error.**
2. **Prepare** – For all static variables memory will be allocated and assigned with **default values.**
3. **Resolve** – All **symbolic memory references** are replaced with the **original references** from **Method Area**.

#### 1.3 Initialization

This is the final phase of Class Loading, here all **static variables** will be assigned with the original values, and the **static block** will be executed.

Scanned by CamScanner

## 2. Runtime Data Area

The Runtime Data Area is divided into 5 major components:

1. **Method Area** – All the **class level data** will be stored here, including **static variables**. There is only one method area per JVM, and it is a shared resource.
2. **Heap Area** – All the **Objects** and their corresponding **instance variables** and **arrays** will be stored here. There is also one Heap Area per JVM. Since the **Method** and **Heap areas** share memory for multiple threads, the data stored is not thread **safe.**
3. **Stack Area** – For every thread, a separate **runtime stack** will be created. For every **method call**, one entry will be made in the stack memory which is called as **Stack Frame**. All **local variables** will be created in the stack memory. The stack area is thread safe since it is not a shared resource. The Stack Frame is divided into three subentities:
    1. **Local Variable Array** – Related to the method how many **local variables** are involved and the corresponding values will be stored here.
    2. **Operand stack** – If any intermediate operation is required to perform, **operand stack** acts as runtime workspace to perform the operation.
    3. **Frame data** – All symbols corresponding to the method is stored here. In the case of any **exception**, the catch block information will be maintained in the frame data.
4. **PC Registers** – Each thread will have separate **PC Registers,** to hold the address of **current executing instruction** once the instruction is executed the PC register will be **updated** with the next instruction.
5. **Native Method stacks** – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

Scanned by CamScanner

## 3. Execution Engine

The bytecode which is assigned to the **Runtime Data Area** will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

1. **Interpreter** – The interpreter interprets the bytecode faster, but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
2. **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
    1. **Intermediate Code generator** – Produces intermediate code
    2. **Code Optimizer** – Responsible for optimizing the intermediate code generated above
    3. **Target Code Generator** – Responsible for Generating Machine Code or Native Code
    4. **Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.
3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling **"System.gc()"**, but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

**Java Native Interface (JNI)**: **JNI** will be interacting with the **Native Method Libraries** and provides the Native Libraries required for the Execution Engine.

**Native Method Libraries**:It is a collection of the Native Libraries which is required for the Execution Engine.

Scanned by CamScanner