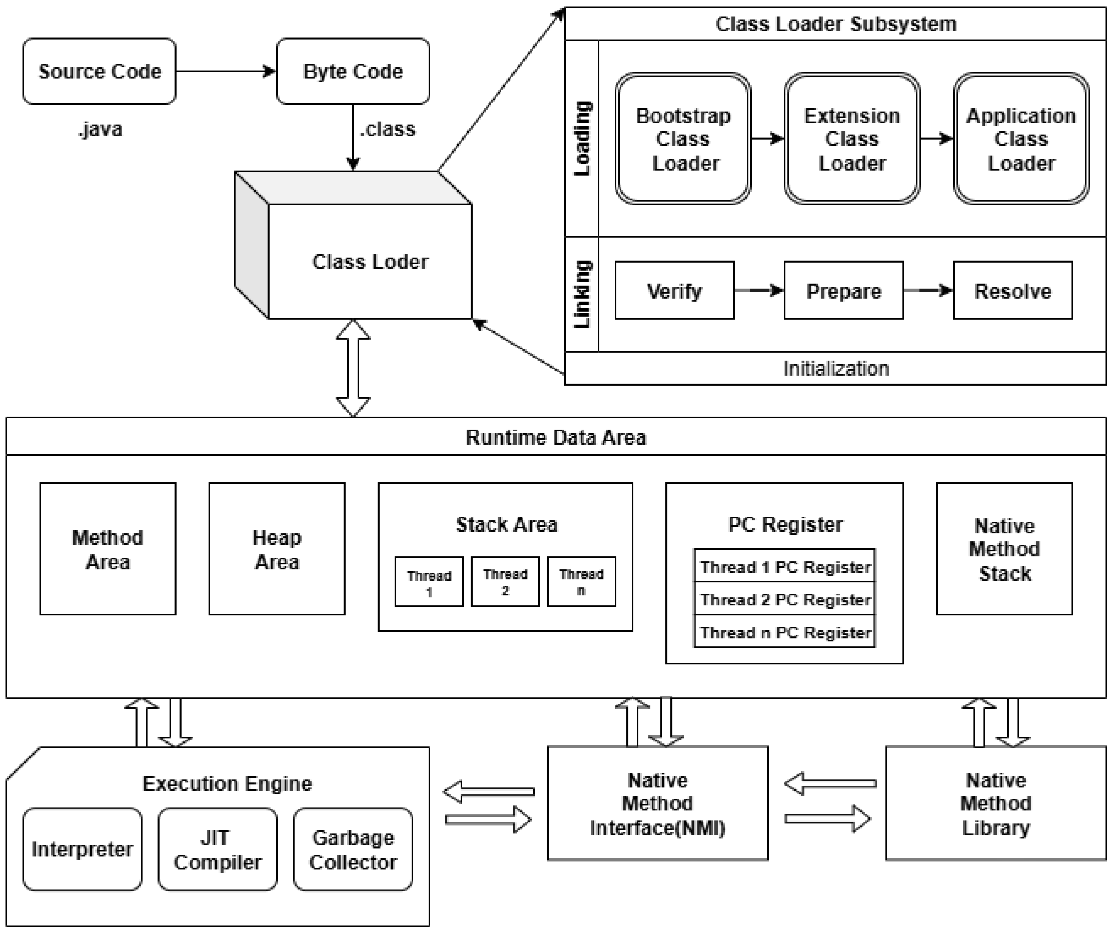
What is JVM ?

JVM (Java Virtual Machine) : It is an engine that provide runtime environment to lunch the Java application and responsible for converting the byte code (.class file) which generated by compiling the (.java file). JVM is a part of Java Runtime Environment(JRE).

JVM Architecture :



JVM is divided into three main subsystems:

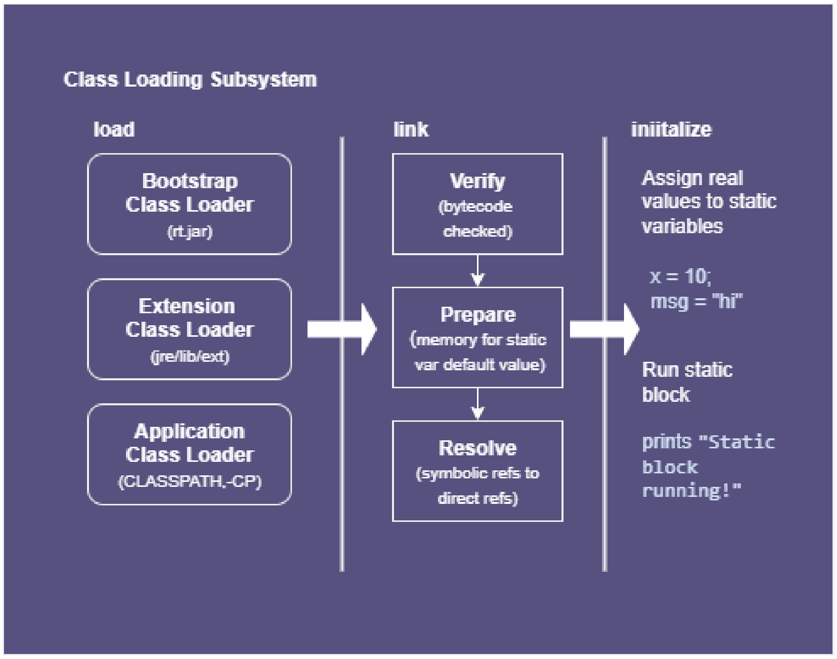
1. ClassLoader Subsystem
2. Runtime Data Area (Memory Area)
3. Execution Engine

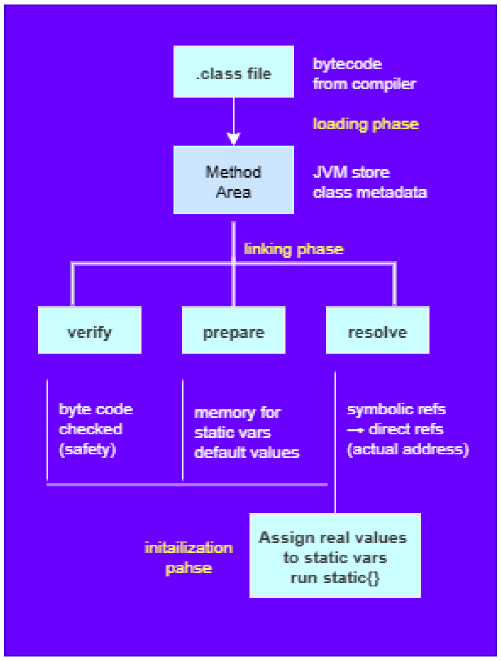
1. ClassLoader Subsystem :

The Class Loader is a part of the Java Runtime Environment (JRE), is responsible for loading class files into the JVM's memory

It have three main phases :

- Loading
- Linking
- Initialization


Class Loading Subsystem
load
link
iniitalize
Bootstrap Class Loader (rt.jar)
Extension Class Loader (jre/lib/ext)
Application Class Loader (CLASSPATH,-CP)
Verify (bytecode checked)
Prepare (memory for static var default value)
Resolve (symbolic refs to direct refs)
Assign real values to static variables
x = 10;
msg = "hi"
Run static block
prints "Static block running!"


.class file
bytecode from compiler
loading phase
Method Area
JVM store class metadata
linking phase
verify
prepare
resolve
byte code checked (safety)
memory for static vars default values
symbolic refs → direct refs (actual address)
initailization pahse
Assign real values to static vars run static{}

Loading : The ClassLoader reads the .class file, generates the corresponding binary data, and saves it in the method area.

There are three types of class loaders:

- **Bootstrap Class Loader**
  Loads core java classes(java.lang, java.util, etc) from rt.jar (or lib directory in JDK/JRE).It’s implemented in native code(C/C++).

- **Extension Class Loader**

It loads classes from the ext folder inside Java installation (JAVA_HOME/lib/ext) or from any folder set in the java.ext.dirs property. Example: if you put mysql-connector.jar inside lib/ext, it will be loaded by the Extension Class Loader.

- **Application Class Loader**
  Loads classes from the classpath (-cp or CLASSPATH environment variable).

Linking : After a class is loaded into memory, the linking phase ensures that the class is ready for use.

It mainly have three phases:

- **Verification**
  Ensure bytecode is valid and doesn't break JVM rules. Prevent illegal memory access or malicious bytecode.

- **Preparation**
  Allocates memory for static fields of the class. Assigns default values.
  Int → 0
  Boolean → false
  Object references → null

  Example:

  ```
  class Test {
     static int x = 10;
     static boolean flag = true;
  }
  ```

  During preparation:
  x = 0
  flag = false

- **Resolution**
  Converts symbolic references (like "java/lang/String") into direct references (actual memory addresses of classes/methods/fields).
  This ensures that when you call a method, the JVM knows exactly where it lives in memory.

Resolution Map:
"java/lang/String" → Class String (Method Area)
"java/io/PrintStream" → Class PrintStream

Initialization :

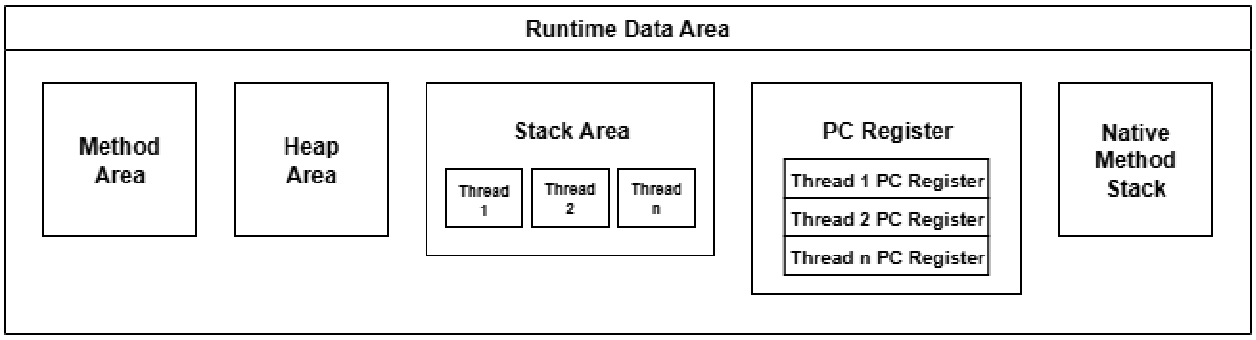
This is the final phase of Class Loading, here all static variables will be assigned with the original values, and the static block will be executed.

x = 10, flag = true.

Static block runs → changes x = 20.

2. Runtime Data Area

The Runtime Data Area is the memory structure created by the JVM when it runs a program. It contains different memory blocks that store class info, objects, methods, variables, and threads.



- **Method Area**
  Stores class metadata (info about class, interfaces, methods, fields).
  Stores static variables and final constants.
  Example: If you declare static int x = 10;, it goes here.

- **Heap Area**
  Stores all objects and their instance variables.

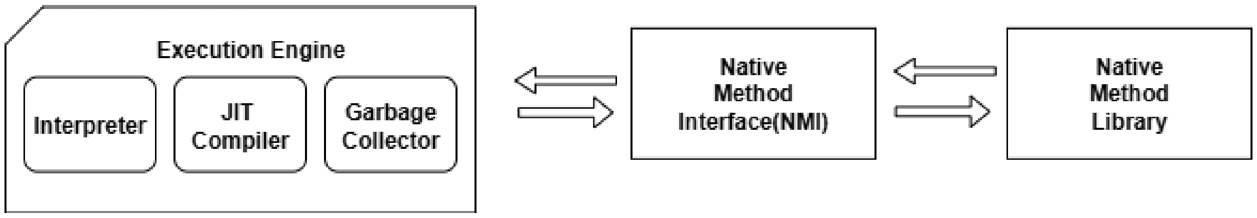
Shared by all threads.
Example: new Student(); → object created in heap.

- **JVM Stack (Per Thread)**
  Each JVM thread has its own JVM stack.
  It stores frames, holds local variables, partial results, references to heap objects.
  Example: In method int sum(int a, int b), a and b are stored here.

- **Program Counter (PC) Register (Per Thread)**
  Each thread has a PC register.
  Stores the address of the current instruction being executed.
  Helps JVM know "what to execute next".

- **Native Method Stack**
  Stores native method calls (written in languages like C/C++).
  Example: If your Java program uses a library written in C, its call goes here.

3. **Execution Engine**
   The Execution Engine (EE) is the heart of JVM.
   After the Class Loader loads classes and the Runtime Data Area allocates memory.
   The Execution Engine executes the bytecode instructions.
   It is responsible for running the program line by line.



- **Interpreter**
  Reads bytecode instructions one by one and executes them.
  Simple, but slower because it interprets repeatedly.

- **JIT (Just-In-Time) Compiler**
  To improve speed, JVM uses JIT compiler.

It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.

- **Garbage Collector (GC)**
  Manages memory automatically.
  Frees memory of objects that are no longer referenced.
  Example: If an object is not used anywhere, GC removes it.

- **Native Method Interface (JNI)**
  Allows Java code to call functions written in other languages like C, C++.
  Example: Java program using a C library for faster image processing.

- **Native Method Libraries**
  It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.