JVM Architecture

Java Virtual Machine (JVM)

---

1. OVERVIEW

What is JVM?

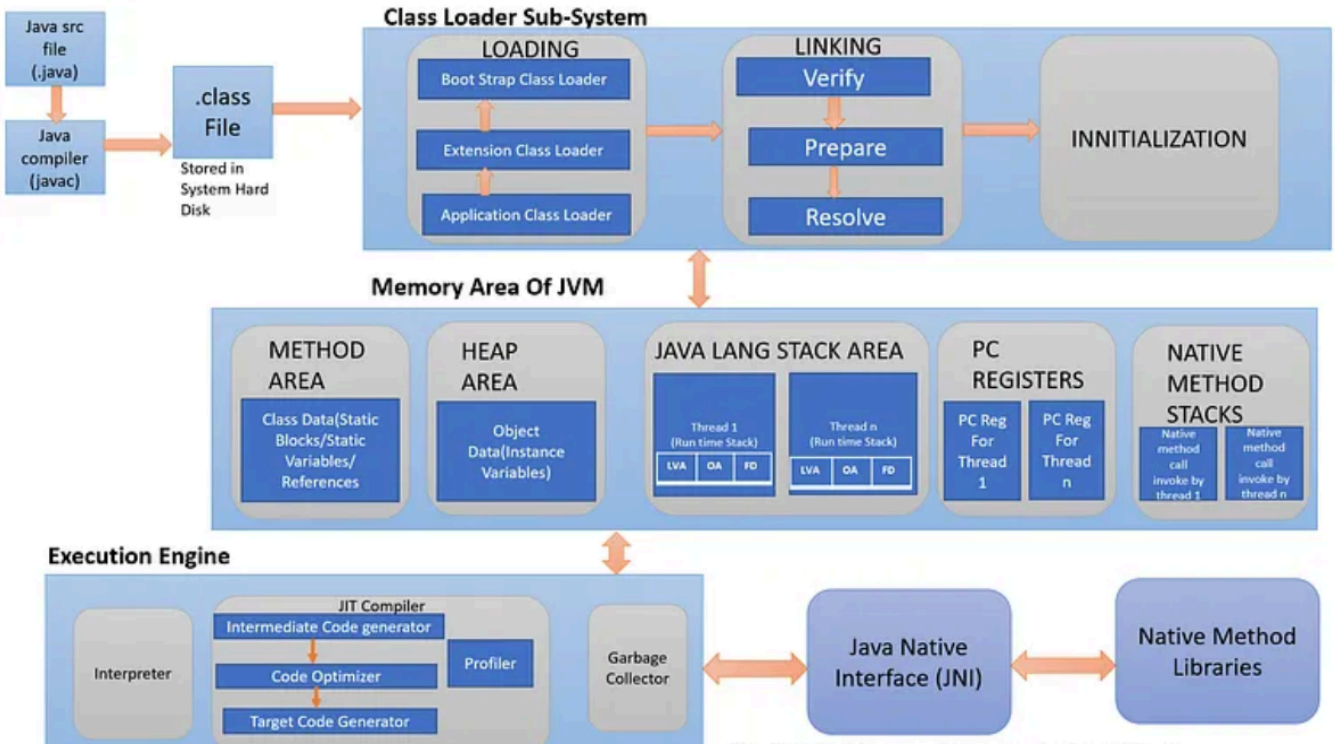
The Java Virtual Machine (JVM) is an abstract computing machine that enables a computer to run Java programs. It's the cornerstone of Java's "Write Once, Run Anywhere" (WORA) principle.

Key Responsibilities

1. **Load** bytecode
2. **Verify** bytecode for security
3. **Execute** bytecode
4. **Provide** runtime environment
5. **Manage** memory (Garbage Collection)

---

2. JVM ARCHITECTURE DIAGRAM



3. DETAILED COMPONENT EXPLANATION

3.1 CLASS LOADER SUBSYSTEM

The class loader is responsible for loading, linking, and initializing Java classes.

A. Loading Phase

Three types of class loaders work hierarchically:

```
Bootstrap ClassLoader (Parent)
        ↓
Extension ClassLoader
        ↓
Application ClassLoader (Child)
```

1. **Bootstrap ClassLoader**
    - Written in native code (C/C++)
    - Loads core Java classes from rt.jar (java.lang, java.util, etc.)
    - No parent
    - Example: String.class, Object.class
2. **Extension ClassLoader**
    - Loads classes from JDK extensions directory (jre/lib/ext)
    - Parent: Bootstrap
    - Example: Security extensions, localization
3. **Application ClassLoader**
    - Loads classes from application classpath
    - Parent: Extension
    - Example: Our Student.class, Main.class

Loading Process:

```
Application CL checks cache
   ↓ (not found)
Delegates to Extension CL
   ↓ (not found)
Delegates to Bootstrap CL
   ↓ (not found)
Bootstrap searches
   ↓ (not found)
Extension searches
   ↓ (not found)
```

Application searches & loads

B. Linking Phase

1. Verification

- Ensures bytecode is valid and secure
- Checks: File format, metadata, bytecode structure, symbolic references
- Example checks:
    - Is it a valid .class file?
    - Does bytecode follow JVM rules?
    - Are there no illegal type casts?

2. Preparation

- Allocates memory for class variables
- Assigns default values

Example:
```
 static int count;  // Allocated, default = 0static final double PI = 3.14;  // Allocated, value = 3.14
```

3. Resolution

- Replaces symbolic references with direct references
- Symbolic: "com/example/Student" → Direct: Memory address

C. Initialization Phase

- Executes static initializers
- Assigns actual values to static variables

Example:
```
 static {    System.out.println("Class loaded!");    count = 100;}
```

-

3.2 RUNTIME DATA AREAS (MEMORY STRUCTURE)

A. METHOD AREA (Shared by all threads)

Stores:

- Class-level data: Class name, parent name, methods, variables
- Static variables
- Constant pool

- Method bytecode

Example:

```
public class Student {
   static int studentCount = 0;  // Stored in Method Area

   public void display() {       // Method bytecode in Method Area
       // ...
   }
}
```

Characteristics:

- Created at JVM startup
- One per JVM
- Garbage collected
- Can throw OutOfMemoryError

B. HEAP (Shared by all threads)

Stores:

- All objects
- Instance variables
- Arrays

Example:

```
Student s1 = new Student();  // Object created in Heap
Student s2 = new Student();  // Another object in Heap
int[] arr = new int[10];     // Array in Heap
```

Garbage Collection Strategy:

- Minor GC: Cleans Young Generation (frequent, fast)
- Major GC: Cleans Old Generation (less frequent, slower)
- Full GC: Cleans entire Heap (rare, slowest)

C. STACK (Per thread)

Stores:

- Method call frames
- Local variables
- Partial results

- Method return values

Example:

```
public void calculate(int a, int b) {  // Frame created
    int sum = a + b;                   // Local variable in frame
    System.out.println(sum);
}  // Frame destroyed
```

Characteristics:

- LIFO (Last In, First Out)
- One per thread
- Fixed size
- Throws StackOverflowError if full

D. PROGRAM COUNTER (PC) REGISTER (Per thread)

Stores:

- Address of current instruction being executed
- Points to next instruction

Example:

```
Instruction 1: iload_1     ← PC points here
Instruction 2: iload_2     ← PC moves here next
Instruction 3: iadd        ← Then here
```

E. NATIVE METHOD STACK (Per thread)

Stores:

- Information for native methods (written in C/C++)
- Similar to Java stack but for native code

3.3 EXECUTION ENGINE

A. INTERPRETER

Function: Executes bytecode line by line

Process:

Bytecode → Read → Interpret → Execute → Next instruction

Advantages:

- Simple
- Starts executing immediately

Disadvantages:

- Slow (interprets same code multiple times)
- Method called 100 times = interpreted 100 times

B. JIT COMPILER (Just-In-Time)

Function: Compiles bytecode to native machine code for frequently used methods

Process:

1. Profiler monitors execution
2. Identifies "hot spots" (frequently executed code)
3. JIT compiles bytecode → native code
4. Caches native code
5. Future calls use cached native code

Components:

1. **Profiler**: Monitors which methods are called frequently
2. **Compiler**: Converts bytecode to native code
3. **Code Cache**: Stores compiled native code

Optimization Techniques:

- Method inlining
- Dead code elimination
- Loop optimization
- Constant folding

Example:

```
// First 100 calls: Interpreted (slow)
for (int i = 0; i < 100; i++) {
    calculate();  // Interpreter
}

// After 100 calls: JIT compiles
// Next calls: Native code (fast)
```

```
for (int i = 0; i < 10000; i++) {
    calculate();  // Native code
}
```

C. GARBAGE COLLECTOR

Function: Automatically manages memory by removing unused objects

Algorithm Steps:

1. **Mark**: Identify reachable objects (starting from GC roots)
2. **Sweep**: Remove unreachable objects
3. **Compact**: Defragment memory (optional)

GC Roots:

- Local variables in method stacks
- Static variables in Method Area
- Active threads
- JNI references

Example:

```
Student s1 = new Student();  // Object 1 created
Student s2 = new Student();  // Object 2 created
s1 = null;                   // Object 1 eligible for GC
// GC will eventually remove Object 1
```

Types of GC:

1. **Serial GC**: Single-threaded
2. **Parallel GC**: Multiple threads for Young Generation
3. **CMS (Concurrent Mark Sweep)**: Low pause times
4. **G1 (Garbage First)**: Large heaps, predictable pause times
5. **ZGC**: Ultra-low latency (Java 11+)

3.4 JAVA NATIVE INTERFACE (JNI)

- Bridge between Java and native applications
- Allows Java code to call C/C++ libraries
- Example: Hardware access, OS-specific operations

3.5 NATIVE METHOD LIBRARIES

- C/C++ libraries required by JVM
- Platform-specific code

---

4. INTERPRETER vs JIT COMPILER

Feature	Interpreter	JIT Compiler
Execution	Line by line	Compiles entire method
Speed	Slower	Faster (after compilation)
Memory	Less memory	More memory (caches code)
Startup	Fast startup	Slower startup
When Used	First execution	After threshold reached
Code Reuse	No	Yes (cached)

Combined Approach (HotSpot JVM):

Start: Interpreter (fast startup)
  ↓
Monitor execution (Profiler)
  ↓
Identify hot methods
  ↓
JIT compile hot methods
  ↓
Use native code for hot methods
Use interpreter for cold methods

Benefits:

- Fast startup (interpreter)
- Fast execution (JIT for hot code)
- Memory efficient (only compile what's needed)

---

5. BYTECODE EXECUTION PROCESS

Step-by-Step Execution

Java Source Code:

```
public class Hello {
   public static void main(String[] args) {
      int a = 5;
      int b = 3;
      int sum = a + b;
      System.out.println(sum);
   }
}
```

Step 1: Compilation (javac)

javac Hello.java → Hello.class (bytecode)

Step 2: Class Loading

1. Application ClassLoader loads Hello.class
2. Verification: Check bytecode validity
3. Preparation: Allocate memory
4. Resolution: Resolve references
5. Initialization: Execute static initializers

Step 3: Execution

Method Area: Stores class metadata **Heap**: (none in this example) **Stack**: Creates frame for main

PC Register: Points to current instruction (0 → 1 → 2...)

Step 4: Interpretation/Compilation

- First run: Interpreter executes bytecode
- If called many times: JIT compiles to native code

---

6. "WRITE ONCE, RUN ANYWHERE" (WORA)

Key Points

1. **Source Code**: Write once in Java
2. **Bytecode**: Compiled to platform-independent format
3. **JVM**: Platform-specific interpreter/compiler
4. **Execution**: JVM handles platform differences

Why WORA?

- Bytecode is abstract (not tied to specific CPU/OS)
- JVM handles translation to native code
- Same .class file runs on any platform with JVM

JVM vs JRE vs JDK

- **JVM**: Virtual machine that executes bytecode
- **JRE**: JVM + libraries to run Java programs
- **JDK**: JRE + development tools (javac, debugger)