Chapter 11

# JVM Architecture

- In this chapter, You will learn
  - Definition of VM and JVM
  - JVM Block diagram
  - Runtime areas
  - Class Loader subsystem
  - Thread and StackFrame architecture
  - Five phase diagram
  - JVM architecture with all types of variables and methods
- By the end of this chapter- you will learn defining, declaring, calling variables, and its execution control flow.

## Interview Questions

**By the end of this chapter you answer all below interview questions**

- Definition of JVM
- JVM Architecture block diagram
- JVM Runtime areas
- What does happen when we run "java" command
- Five phases in program compilation and execution
- ClassLoader sub system architecture
- Thread architecture
- Stack Frame architecture
- Program execution process in stack
- JVM architecture with all types of variables and method execution

# JVM Architecture

In this chapter I presented only the essential information. To get more detailed information on JVM architecture refer "*SUN specification*" or a text book "*Inside the JVM*, by Bill Venners".

The source of this chapter is "*Inside the JVM*, by Bill Venners".

**Definition of JVM**

**Virtual Machine**
In general terms VM is a SW that creates an environment between the computer platform and end user in which end user can operate programs.

**Original meaning for VM**
As per its functionality is Creation of number of different identical execution environments on a single computer to execute programs is called VM.

**Java Virtual Machine**
It is also a VM that runs Java bytecode by creating five identical runtime areas to execute class members. This bytecode is generated by java compiler in a JVM understandable format.

**Q) How can we start JVM process?**
A) By using "***java***" tool.

The Java launcher, *java*, initiates the Java virtual machine instance.

**Types of JVMs**:
The Java 2 SDK, Standard Edition, contains two implementations of the Java virtual machine

- Java HotSpot Client VM
- Java HotSpot Server VM

**Java HotSpot Client VM:**
The Java HotSpot Client VM is the default virtual machine of the Java 2 SDK and Java 2 Runtime Environment. As its name implies, it is tuned for best performance when running applications in a client environment by reducing application start-up time and memory footprint.

**Java HotSpot Server VM:**
The Java HotSpot Server VM is designed for maximum program execution speed for applications running in a server environment. The Java HotSpot Server VM is invoked by using the *-server* command-line option when launching an application, as in

```
java -server MyApp
```

Some of the features Java HotSpot technology, common to both VM implementations, are the following.

**Adaptive compiler**

- Applications are launched using a standard interpreter, but the code is then analyzed as it runs to detect performance bottlenecks, or "hot spots".
- The Java HotSpot VMs compile those performance-critical portions of the code for a boost in performance, while avoiding unnecessary compilation of seldom-used code (most of the program).
- The Java HotSpot VMs also uses the adaptive compiler to decide, on the fly, how best to optimize compiled code with techniques such as in-lining.
- The runtime analysis performed by the compiler allows it to eliminate guesswork in determining which optimizations will yield the largest performance benefit.

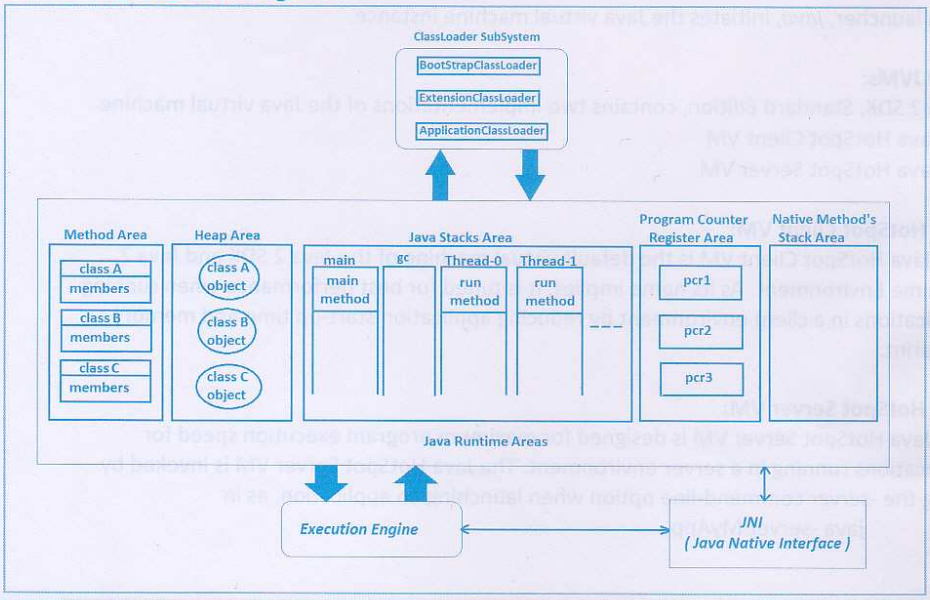
**Rapid memory allocation and garbage collection:**

- Java HotSpot technology provides for rapid memory allocation for objects, and it has a fast, efficient, state-of-the-art garbage collector.

**Thread synchronization:**

- The Java programming language allows for use of multiple, concurrent paths of program execution (called "threads").
- Java HotSpot technology provides a thread-handling capability that is designed to scale readily for use in large, shared-memory multiprocessor servers.

**JVM Architecture block diagram**



**Explanation on Runtime Areas**

Whenever we execute a class by specifying its corresponding class name by using the command "*java <ClassName>*", the Java launcher, **java**, immediately initiates the Java Runtime environment for the class execution as a layer on top of OS, and further the entire setup is divided in to 5 Java Runtime Areas named as

1. Method Area
2. Heap Area
3. Java Stacks Area
4. Program counter registers area
5. Native Methods Stacks area

**Method Area**

- All classes' bytecode is loaded and stored in this runtime area, and all static variables are created in this runtime area.

**Heap Area**

- It is the main memory of JVM. All *objects* of classes - *non-static variables memory*- are created in this runtime area. This runtime area memory is a finite memory.
- This area can be configured at the time of setting up of runtime environment using non standard option like

  java ***-xms <size>*** classname
- This area can be expandable by its own, depending on the objects creation.
- *Method area and Heap area both are sharable memory areas.*

**Java Stacks area**

- In this runtime area all Java methods are executed.
- In this runtime JVM by default creates two threads, they are
  - main thread
  - garbage collector thread
- Main thread is responsible to execute Java methods stats with main method, also responsible to create objects in heap area if it finds "new" keyword in any method logic
- Garbage collector thread is responsible to destroy all unused objects from heap area.

  ***Note:*** Like in C++, in Java, we do not have destructors to destroy objects.
- For each method execution JVM creates separate block in main thread. Technically this block is called Stack Frame. This stack frame is created when method is called and is destroyed after method execution.

  ***Note:*** Java operations are called "stack based operations (sequential)", because every method is executed only in stack.

**Program Counter Registers Area**

- In this runtime area, a separate program counter register is created for every thread for tracking that thread execution by storing its instruction address.

**Native Methods Stacks Area**

- In Native Methods stack area, all Java native methods are executed.

**Q) What is a native method?**
The Java method that has logic in C, C++ is called native method. To create native method, we must place native keyword in its prototype and it should not have body.

***For example***

```
class Example{
        public static native int add(int x, int y);

        public static void main(String[] args) {
                add(10, 20);
        }
}
```

The above program compiled fine, but in execution it leads RE: ***java.lang.UnsatisfiedLinkError.***
We have defined native method, but we have not defined it required C program and not linked.

**Q) Suppose if we provide C, or C++ program for the above native method, who will take care of linking this java native method prototype to original native definition?**
A) *JNI* - Java Native Interface - is a mediator between Java native method and original native method for linking its method calls.

**Execution engine**

- All executions happening in JVM are controlled by Execution Engine.

**ClassLoader subsystem**
ClassLoader is a class that is responsible to load classes into JVM's method area.

We have basically three types of class loaders

1. ApplicationClassLoader
2. ExtensionClassLoader
3. BootstrapClassLoader

- *ApplicationClassLoader* is responsible to load classes from Application Classpath, (current working directory). It basically uses Classpath environment variable to locate the class's ".class" file.
- *ExtensionClassLoader* is responsible to load classes from Extension Classpath, ("%JAVA_HOME%\jre\lib\ext" folder)

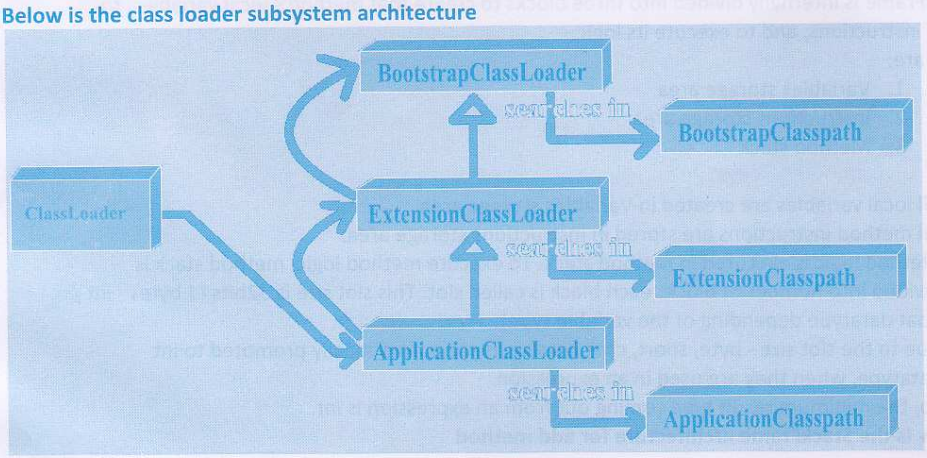
- *BootStrapClassLoader* is responsible to load classes from BootStrap Classpath (%JAVA_HOME%\jre\lib\rt.jar). These classes are predefined classes.

**ClassLoader Working procedure:**

- When JVM come across a type, it check for that class bytecodes in method area.
- If it is already loaded it makes use of that type.

- If it is not yet loaded, it requests class loader subsystem to load that class's bytecodes in method area form that class respective Classpath.
- Then ClassLoader subsystem, first handovers this request to *ApplicationClassLoader*, then application loader will search for that class in the folders configured in *Classpath* environment variable
- If class is not found, it forwards this request to *ExtenstionClassLoader*. Then it searches that class in *ExtensionClasspath*.
- If class is not found, it forwards this request to *BootStrapClassLoader*. Then it searches that class in *BootStrapClasspath*.
- If here also class not found, JVM throws an exception "*java.lang.NoClassDefFoundError*" or "*java.lang.ClassNotFoundExcetion*"
- If class is found in any one of the Classpaths, the respective ClassLoader loads that class into JVM's method area.
- Then JVM uses that loaded class bytecodes to complete the execution.

**Below is the class loader subsystem architecture**




**Thread & StackFrame Architecture**

- thread is an independent sequential flow of execution created in JSA.
- StackFrame is a sub block created inside a thread for executing a method or block, and is destroyed automatically after the completion of that method execution.
- If this method has any local variables, they are all created inside that method's stack frame, and are destroyed automatically when StackFrame is destroyed.

**Below architecture shows main thread with stack frames**

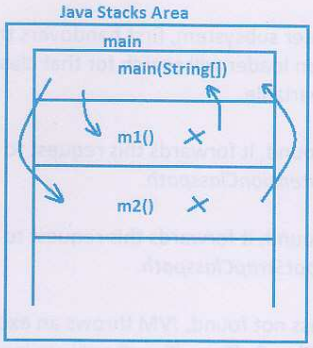
```
class Example{

        static void m1(){
                System.out.println("m1");
        }

        static void m2(){
                System.out.println("m1");
        }

        static void m3(){
                System.out.println("m1");
        }

        public static void main(String[] args){
                System.out.println("main");
                m1();
                m2();
        }
}
```




**StackFrame architecture**

StackFrame is internally divided into three blocks to create that method's local variables, to store instructions, and to execute its logic.

They are:

1. Variables storage area
2. Instructions storage area
3. Method stack

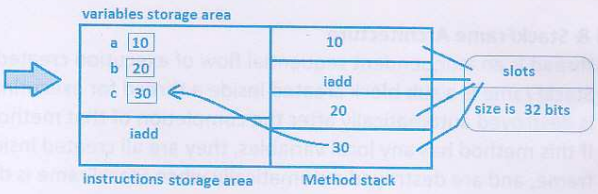
- All local variables are created in Variables storage area
- All method instructions are stored in Instructions storage area.
- Method logic is executed in method stack. To execute method logic, method stack is divided into number of blocks each block is called slot. This slot size is 32bits (4 bytes - int / float datatype depending of the variable type).
- Due to the slot size - byte, short, char datatypes are automatically promoted to int datatype, when they are used in an expression.
- So, the minimum result type coming out from an expression is *int*.

**Below is the StackFrame architecture for add method**

```
static void add(){

        int a = 10;
        int b = 20;
        int c = a + b;
        System.out.println("Result: "+c);

}
```



**Draw JVM architecture for the below program**

```
class Example{
        static int a = 10;
        static int b = 20;

                int x = 30;
                int y = 40;

        public static void main(String[] args){
                int p = 50;
                int q = 60;

                Example e = new Example();
        }
}
```

**JVM Architecture final diagram with conclusions**

- Complete class bytecodes are stored in method area with *java.lang.Class* object and all static variables get memory in method area
- All non-static variables i.e. objects, are created in Heap area when object is created.
- All Java methods, blocks and constructors are executed in Java stacks area in main thread by creating separate stack frame.
- So, all local variables are created in its method's stack frame.

**What happened in JVM when we execute java command?**

- When java command is executed, JVM is created as a layer on top of OS, and is divided in 5 runtime areas as shown above.
- For executing the requested classes, JVM internally performs below three phases.

They are:

**1. Classloading:**

JVM requests class loader to load the class from its respective Classpath. Then class is loaded in method area by using java.lang.Class object memory.

**2. Bytecode verification phase:**

After Classloading, BytecodeVerifier verifies the loaded bytecode's internal format. If those bytecodes are not in the JVM understandable format, it terminates execution process by throwing exception "*java.lang.ClassFormatError*". If loaded bytecodes are valid, it allows interpreter to execute those bytecodes

**3. Execution / interpretation phase:**

Then interpreter converts bytecodes into current OS understandable format.

Finally, JVM generates output with the help of OS.