

# Java Virtual Machine (JVM) Delving Deep into its Architecture

A Java Virtual Machine can be thought of as an abstract computer that is defined by certain specifications. The author leads readers deep into the architectural details of JVM to give them a better grasp of its concepts.



**A** virtual machine, or virtualisation, has emerged as a key concept in operating systems. When it comes to application programming using Java, platform-independent value addition is possible because of its ability to work across different operating systems. The Java Virtual Machine (JVM) plays a central role in making this happen. In this article, let us delve deep into the architectural details of JVM to understand it better.

## Building basics

Let us build our basics by comparing C++ and a Java program with a simple diagram (Figure 1). The C++ compiled object code is OS-specific, say an x86-based Windows machine. During execution, it will require a similar OS, failing which the program will not run as expected. This makes languages like C++ platform- (or OS) dependent. In contrast, Java compilation produces platform-independent byte code, which will get executed using the native JVM. Because of this fundamental difference, Java becomes platform-independent, powered by JVM.

## Exploring JVM architecture

Fundamentally, the JVM is placed above the platform and below the Java application (Figure 2).

Going further down, the JVM architecture pans out as shown in Figure 3. Now let us look into each of the blocks in detail.

In a nutshell, JVM architecture can be divided into two different categories, the details of which are provided below.

**Class loader subsystem:** When the JVM is started, three class loaders are used.

- System class loader:* System class loader maps the class-path environment variables to load the byte code.
- Extension class loader:* Extension class loader loads the byte code from *jre/lib/ext*.
- Bootstrap class loader:* The bootstrap class loader loads the byte code from *jre/lib*.

**Method area:** The method area (or class area) stores the structure of the class once it is loaded by the class loader. The method area is very important; it does two things once the class is stored in this area:

- Identification:* All static members (variable, block, method, etc) are identified from top to bottom.
- Execution:* Static variables and static blocks are executed after the identification phase, and static methods are executed when they are called out. Once all static variables and blocks are executed, only then will the static method be executed.

**Heap area:** The heap area stores the object. Object instances are created in this area. When a class has instance members (instance variable, instance method and instance block), these members are identified and executed only when the instance is created at the heap area.

## Java stacks

In the Java stack area, two threads (main thread and garbage collector) are always running. When the user creates any new thread, it becomes the third thread (Thread-0). When the user creates any method, it is executed by the main thread, inside a stack frame (Figure 4). Each method gets its own stack frame to execute. The stack frame has three sections – the local variable

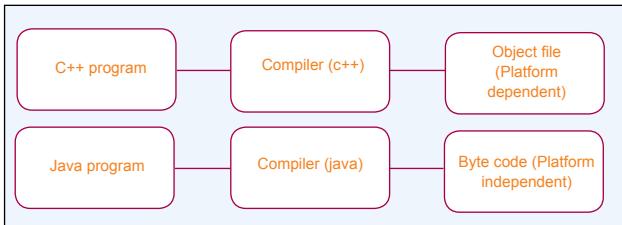


Figure 1: Differences in C++ and Java compilation

storage section, the instruction storage section and memory slots to perform operations. Each memory slot inside the stack frame is, by default, of 4 bytes, but according to the size of the variable, the size of the slot also shrinks or expands.

According to Figure 4, the addition of a 2-byte variable (a and b) will not result in a byte because the default size of the stack frame memory slot is 4 bytes, which can't be inserted into the byte (r) variable; so we need to typecast it as (r = (byte)a+b).

**The PC register:** The program counter (PC) register contains the address of the Java virtual machine instruction currently being executed.

**Native method stacks:** All the native methods are executed in this area.

**Execution engine:** All executions happening in JVM are controlled by the execution engine.

**Native method interface:** Java Native Interface (JNI) enables the Java code running in JVM to call and be called by the native application and libraries (Native Method Libraries) written in other languages such as C and C++.

## JVM in action

Now let us take a look at a few Java code snippets and understand the role of various JVM components, during execution.

**Example 1 (when all members are static):** To understand this example, the method area is explained earlier in this article. According to the method area, all static members of a class are identified and executed in the same order in which they appear. When all static members are executed, only then is the main function executed.

```

class Test{
    static int a = m1();
    static{
        System.out.println ("in static block");
    }
    public static int m1(){
        System.out.println("in m1");
        return 10;
    }
    public static void main(String[] args){
        System.out.println("in main");
    }
}

```

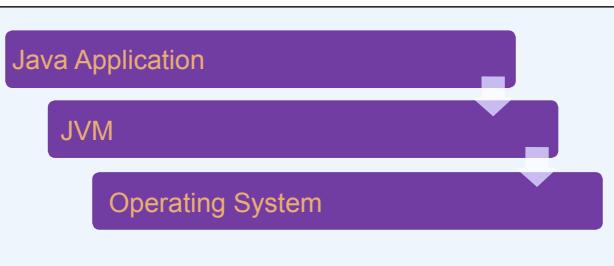


Figure 2: How JVM fits between the OS and Java application

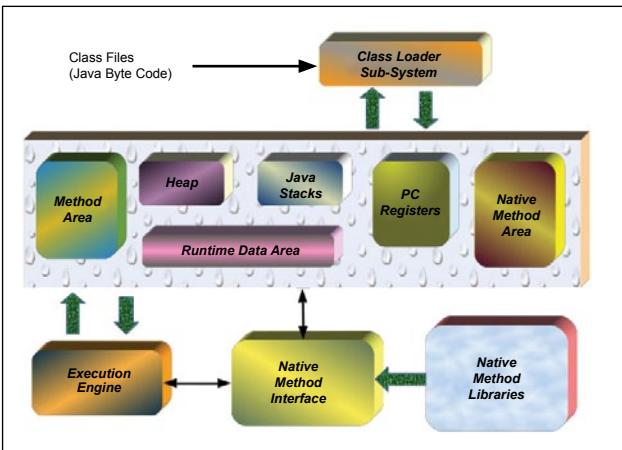


Figure 3: Architecture of JVM

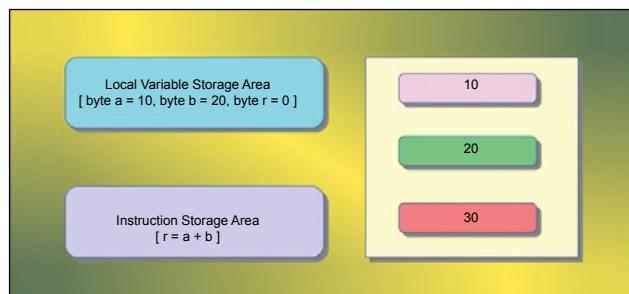


Figure 4: Architecture of the stack frame

}

The output of the above program is:

```

in m1
in static block
in main

```

**Example 2 (in case of inheritance):** To understand this example, you need to understand the method area as well as Example 1. In this example, we are trying to explain that, when a class is inherited from any other class, the static members are identified and executed from the top to the bottom or from parent class to child class, and the main function will be executed at the end.

```

class Vehicle{
    static int a = m1();
    public static int m1(){

```

```

        System.out.println("in m1");
        return 10;
    }
    static{
        System.out.println("Vehicle static block");
    }
}
class Car extends Vehicle{

    static int b = m2();
    public static int m2(){
        System.out.println("in m2");
        return 10;
    }
    static{
        System.out.println("Car static block");
    }
    public static void main(String[] args){
        System.out.println("in main");
    }
}

```

The output of the above code is:

```

in m1
Vehicle static block
In m2
Car static block
in main

```

The above output clarifies one thing—that the order of identification and the execution of static members in case of inheritance will occur from top to bottom or from parent class to child class.

**Example 3 (when members are static, non-static and constructor):** In this example, the effect of the heap area on the Java program is explained (the heap area itself has been covered earlier in the article). All the Java objects and instance members are initialised here. The method area is about static members and the heap area is about object and non-static members. In the heap area, when the Java objects are created, only then is the instance variable identified and executed. The constructors are executed in the end.

```

class Test{
    static int a=m1();
    int b=m2();
    {
        System.out.println("instance block");
    }
    public int m2(){
        System.out.println("in m2");
        return 10;
    }
}

```

```

static {
    System.out.println("in static block");
}
public static int m1(){
    System.out.println("in m1");
    return 15;
}
public Test(){
    System.out.println("in constructor");
}
public static void main(String[] args){
    System.out.println("in main");
    Test t = new Test();
}

```

The output of the above code is:

```

in m1
in static block
in main
in m2
instance block
in constructor

```

The above output clarifies the following three points:

1. The static member is both identified as well as executed first.
2. Instance members (non-static variables and non-static methods) are identified and executed only if the instance is created, and the order of identification and execution will be from top to bottom.
3. Constructors are always executed last.

The Java programming language becomes platform independent because of JVM and the nature of its byte code. Because of this, Java has scaled across multiple machines, platforms and devices, powering enterprise class Web applications as well as consumer mobile applications. The architecture of JVM has a significant effect on a Java program. Both static and non-static members of a Java class are treated differently by JVM (examples 1 and 3). Static members of a class are identified and executed as soon as the class is loaded. Non-static members are identified and executed only if the instance of the class is created. 

### By: Vikas Kumar Gautam

The author is a mentor at Emertxe Information Technology (P) Ltd. His main areas of expertise include application development using Java/J2EE and Android for both Web and mobile devices. A Sun Certified Java Professional (SCJP), his interests include acquiring greater expertise in the application space by learning from the latest happenings in the industry. He can be reached at vikash\_kumar@emertxe.com