**JVM Architecture – Understanding JVM Internals**

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 we should know the Architecture of JVM, this enables us to write code more efficiently. In this JVM architecture in Java with diagram article, we will learn more deeply about **JVM architecture** in Java and **different components** of a JVM.

**What is a JVM in Java ?**

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** will be compiling the java file into a java **.class** file.  The **.class** file is input to JVM which Loads and executes the class file. Below goes the Architecture of JVM.

**JVM Architecture Diagram**

**[](https://javainterviewpoint-7ac9.kxcdn.com/wp-content/uploads/2016/01/JVM-Architecture.png)**

**How JVM works in Java ?**

As shown in the above architecture diagram 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 when it refers to a class for the first time at **runtime**, not at **compile-time.**It performs three major functionality such as Loading, Linking, and Initialization.

**1.1 Loading**

Classes will be loaded by this component. BootStrap ClassLoader, Extension ClassLoader, Application ClassLoader are the three class loader which will help in achieving it.

1. **BootStrap 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 **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 variable**](https://www.javainterviewpoint.com/use-of-static-keyword-in-java/)will be assigned with the original values and [**static block**](https://www.javainterviewpoint.com/java-static-import/) will be executed.

**2. Runtime Data Area**

Runtime Data Area is divided into 5 major components

1. **Method Area** – All the **Class level data** will be stored here including **static variables**. **Method Area** is **one per JVM** and it is a shared resource.
2. **Heap Area** – All the **Objects** and its corresponding**instance variables** and **arrays** will be stored here. **Heap Area** is also **one per JVM**since **Method area** and **Heap area** shares 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. Stack area is thread safe since it is not a shared resource. Stack Frame is divided into three sub-entities such as
   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** act 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 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, separate native method stack will be created.

**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 byte code and executes one by one.

1. **Interpreter** – Reads the bytecode, interprets it and executes it one by one. The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method called multiple times, every time interpretation is required.
2. **JIT Compiler** – JIT Compiler neutralizes the disadvantage of the Interpreter ( a single method called multiple times, each time interpretation is required ), The Execution Engine will be using the help of Interpreter in converting but when it found repeated code it uses 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** – Code Optimizer is responsible for optimizing the intermediate code generated above
   3. **Target Code Generator** – Target Code Generator is responsible for Generating Machine Code/ Native Code
   4. **Profiler** – **Profiler** is a special component, it is responsible for finding the hotspots (i.e) Used to identify whether the method is called multiple time or not.
3. **Garbage Collector** : Garbage Collector is a part of Execution Engine, it collects/removes the unreferenced objects. Garbage Collection can be triggered by calling ***“System.gc()”***, but the execution is not guaranteed. Garbage collector of JVM collects only those objects that are created by **new** keyword. So if you have created any object without **new**, you can use **finalize method** to perform cleanup.

**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.

## ****Fail-Fast Iterator****

As the name sounds the Iterator will fail as soon as the it encounters a change in the collection. What ever the change it may be adding, update or removal of any object in the collection will throw the ConcurrentModificationException

## ****Fail-Safe Iterator****

Whereas the fail-safe iterator will not throw any exception when the collection such as **CopyOnWriteArrayList and ConcurrentHashMap** is modified. As it iterates on the copy of the collection.

**Sorting:**

**Comparable interface:**

The Comparable interface, in the java.lang package, is for when a class has a natural

ordering. Given a collection of objects of the same type, the interface allows you to order the

collection into that natural ordering.

The compareTo() method compares the current instance with an element passed in as an

argument. If the current instance comes before the argument in the ordering, a negative

value is returned. If the current instance comes after, then a positive value is returned.

Otherwise, zero is returned.

 Sorts the specified list into ascending order, according to the natural ordering of its elements.

**Comparator interface:**

Sorts the specified list according to the order induced by the specified comparator

The return values of the compare() method of Comparator are similar to the

compareTo() method of Comparable

**Hashing:**

Hashing is the process of generating a key value (in this case, typically a 32 or 64 bit integer) from a piece of data.This hash value then becomes a basis for organizing and sorting the data. The hash value might be the first n bits of data, the last n bits of data, a modulus of the value, or in some cases, a more complicated function. Using the hash

value, different "hash buckets" can be set up to store data. If the hash values are distributed evenly (which is the case for an ideal hash algorithm), then the buckets will tend to fill up evenly, and in many cases, most buckets will have no

more than one or only a few objects in them. This makes the search even faster.

**Connection Pool:**

A connection pool operates by performing the work of creating connections ahead of time, In the case of a JDBC connection pool, a pool of Connection objects is created at the time the application server (or some other server) starts. These objects are then managed by a pool manager that disperses connections as they are requested by clients and returns them to the pool when it determines the client is finished with the Connection object. A great deal of housekeeping is involved in managing these connections.

When the connection pool server starts, it creates a predetermined number of Connection objects. A client application would then perform a JNDI lookup to retrieve a reference to a DataSource object that implements the ConnectionPoolDataSource interface. The client application would not need make any special provisions to use

the pooled data source;the code would be no different from code written for a nonpooled DataSource.