

The JVM Architecture Explained



by Jackson Joseraj · Sep. 02, 16 · Java Zone

Like (67) Comment (11) Save Tweet

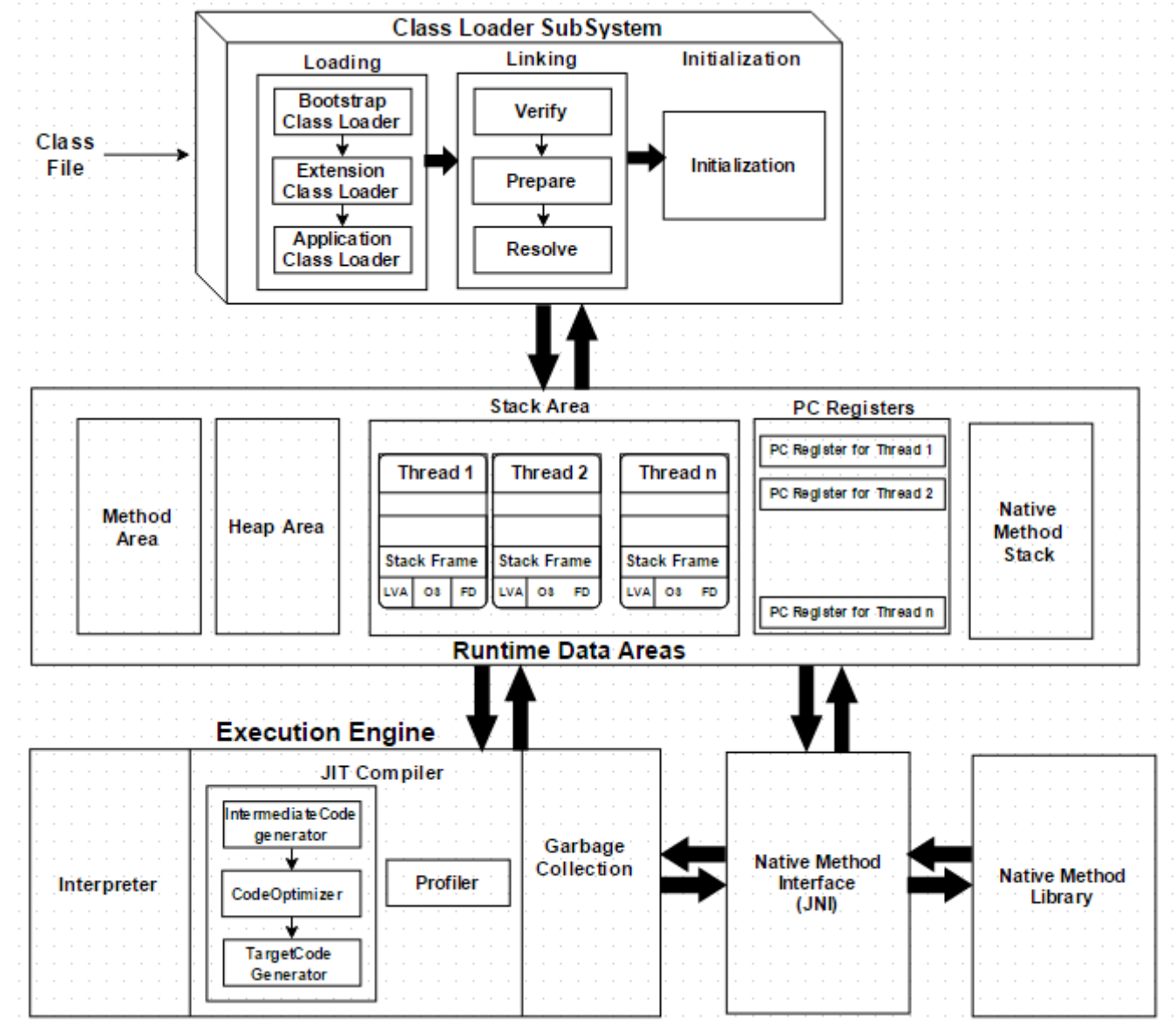
Learn more about how the [Java language, tools and frameworks](#) have been the foundation of countless enterprise systems, brought to you in partnership with [Salesforce](#).

Every Java developer knows that byte code 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 analyses the byte code, interprets the code, and executes it. It is very important as a developer that we should know the Architecture of the JVM, as it enables us to write code more efficiently. In this article we will learn more deeply about the **JVM architecture** in Java and the **different components** of the JVM.

What is the JVM?

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** compiles the Java file into a **Java.class** file, then that **.class** file is input into the JVM, which Loads and executes the class file. Below is a diagram of the Architecture of the JVM.

JVM Architecture Diagram



How Does the JVM Work?

As show in the above architecture diagram, the 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 file when it refers to a class for the first time at **runtime**, not **compile time**.

1.1 Loading

Classes will be loaded by this component. Boot Strap class Loader, Extension class Loader, and Application class Loader are the three class loader which will help in achieving it.

1. **Boot Strap class Loader** – Responsible for loading classes from bootstrap class path, nothing but **rt.jar**. Highest priority will be given to this loader.
2. **Extension class Loader** – Responsible for loading classes which is inside **ext** folder (**jre\lib**).
3. **Application class Loader** – 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 byte code 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 variables** will be assigned with the original values, and **static block** will be executed.

2. Runtime Data Area

The Runtime Data Area is divided into 5 major components:

1. **Method Area** – All the **class level data** will be stored here, including **static variables**. There is only one method area per JVM, and it is a shared resource.
2. **Heap Area** – All the **Objects** and their corresponding **instance variables** and **arrays** will be stored here. There is also one Heap Area per JVM. Since the **Method** and **Heap areas** share 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. The stack area is thread safe, since it is not a shared resource. The Stack Frame is divided into three subentities:

1. **Local Variable Array** – Related to the method how many **local variable** are involved and the corresponding values will be stored here.
2. **Operand stack** – If any intermediate operation is required to perform, **operand stack** acts as runtime workspace to perform the operation.
3. **Frame data** – All symbols corresponding to the method is stored here. In 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, a separate native method stack will be created.

3. Execution Engine

The byte code which is assigned to the **Runtime Data Area** will be executed by the Execution Engine. The Execution Engine reads the byte code and executes it piece by piece.

1. **Interpreter** – The interpreter interprets the byte code faster, but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
2. **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the 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** – Responsible for optimizing the intermediate code generated above
 3. **Target Code Generator** – Responsible for Generating Machine Code or Native Code

4. **Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.

3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling "**System.gc()**", but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

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.

Discover how the [Force.com Web Services Connector \(WSC\)](#) is a code-generation tool and runtime library for use with Force.com Web services, brought to you in partnership with [Salesforce](#).

Topics: JVM