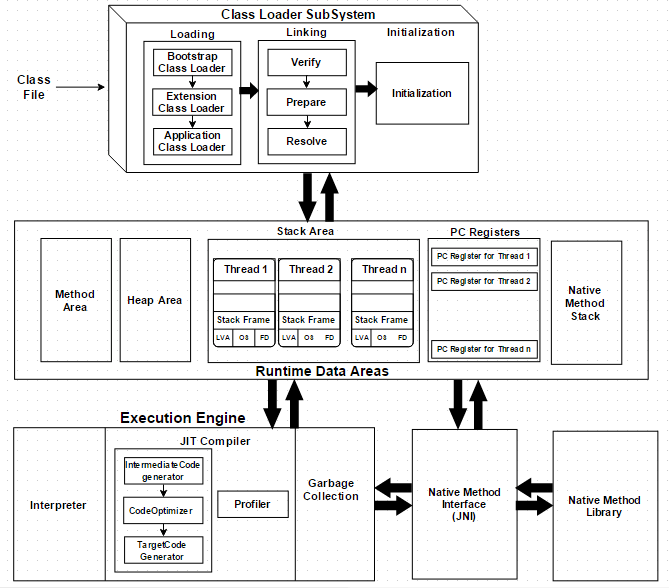
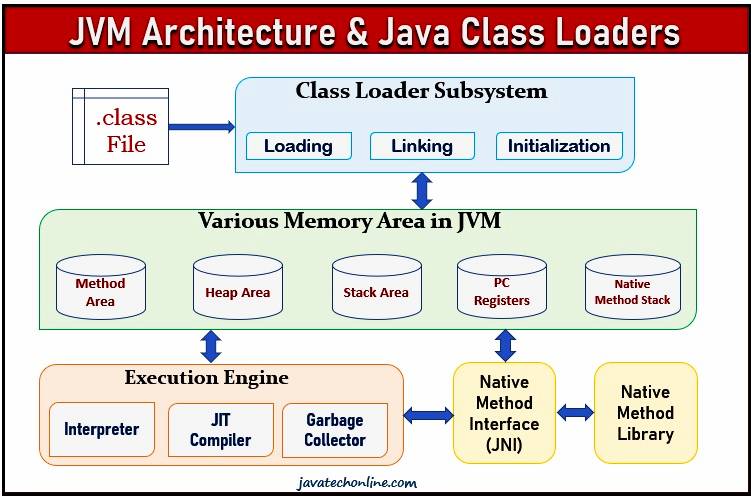
JVM Architecture – 2024



**Simplified Diagram**



There are mainly three sub systems in the JVM as shown in the above diagram,

1. ClassLoader
2. Runtime Memory/Data Areas
3. Execution Engine

**ClassLoader**

This is responsible for loading the classes and it performs three functions such as loading, linking, and initialization.

**Loading**

This process usually starts with loading the main class (class with the main() method). ClassLoader reads the .class file and then the JVM stores the following information in the method area.

* The fully qualified name of the loaded class
* variable information
* immediate parent information
* whether it is a class or interface or enum

**Linking**

This process can be divided into three main parts and they are,

1. **Verification(Verify)**: Check the correctness of the .class file. Byte code verifier will check the followings,

* whether it is coming from a valid compiler or not
* whether the code has a correct structure and format.

If it is not proper then JVM will throw “java.lang.VerifyError” Exception.

2. **Preparation(Prepare)**

In this phase, For all static variables memory will be allocated and assigned with default values based on the data types.

## **3. Resolution(Resolve)**

This is the process of **replacing the symbolic references with direct references** and it is done by searching into the method area to locate the referenced entity. The machine does not understand the name that we give to create objects. So the **JVM will assign memory location for those objects by replacing their symbolic link**s with direct links. **All symbolic memory references are replaced with the original references from Method Area**.

**Initialization**

In this phase, the original values will be assigned back to the static variables as mentioned in the code and a static block will be executed(in any). The execution takes place from top to bottom in a class and from parent to child in the class hierarchy. Most importantly, JVM has a rule saying that the initialization process must be done beforea class becomes an **active use. This is the final phase of ClassLoading; here, all**[**static variables**](http://www.javainterviewpoint.com/use-of-static-keyword-in-java/)**will be assigned with the original values, and the**[**static block**](http://www.javainterviewpoint.com/java-static-import/)**will be executed. Initialization involves executing the initialization code of the class or interface. This can include calling the class’s constructor, executing the static block, and assigning values to all the static variables.**

**Note**: While loading, linking and initialization if any error occurs, then we will get a run time exception saying **java.lang.linkageError**.

**Runtime Data Area**

JVM memory is basically divided into five following parts.

A diagram of a stack area

Description automatically generated

**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. This is where the class data is stored during the execution of the code and this holds the information of static variables, static methods, static blocks, instance methods, class name, and immediate parent class name(if any). Also Constant pools will be saved inside method area.

**Heap Area**

This is where the information of all objects is stored and it’s a shared resource just like the method area. All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM.

**Stack Area**

All the local variables, method calls, and partial results of a program (not a native method) are stored in the stack area. For every thread, a runtime stack will be created. A block of the stack area is known as “**Stack Frame**” and it holds the local variables of method calls. So whenever the method invocation is completed, the frame will be removed (POP). Since this is a stack, it uses a Last-In-First-Out structure. The stack area is thread-safe since it is not a shared resource.

The Stack Frame is divided into three sub-entities:

1. **Local Variable Array** – Related to the method how many local variables are involved and the corresponding values will be stored here. It contains values of local variables & method parameters.
2. **Operand stack** – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation. VM uses it as workspace, some instruction push the values to it & some pop from it & some other to performs arithmetic operations.
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. It contains all symbolic references related to the method. It also contains reference of exception related to method.

**PC Register (Program Counter Register)**

This will hold the thread’s executing information. Each thread has its own PC registers to hold the address of the current executing information and it will be updated with the next execution once the current execution finishes. In brief PC register contains address of currently executing threads.

**Native Method Area**

This will hold the information about the native methods and these methods are written in a language other than Java, such as C/C++. Just like stack and PC register, a separate native method stack will be created for every new thread. All native method calls invoked by the thread will be stored in the corresponding Native method stack.

**Execution Engine**

This is where the execution of bytecode (.class) occurs and it executes the bytecode line-by-line. Before running the program, the bytecode should be converted into machine code.

**Java Native Interface (JNI)**

This is used to interact with the Native(non-java) Method libraries (C/C++) required for the execution. This will allows JVM to call those libraries to overcome the performance constraints and memory management in Java. This acts as a mediator for java method calls and corresponding native libraries. JNI acts as mediator between java method calls & corresponding native libraries.

**Native Method Libraries**

This is a collection of the Native Libraries (written in non-java language in the form of .dll or .so file extension), which is required for the Execution Engine and be accessed through JNI.