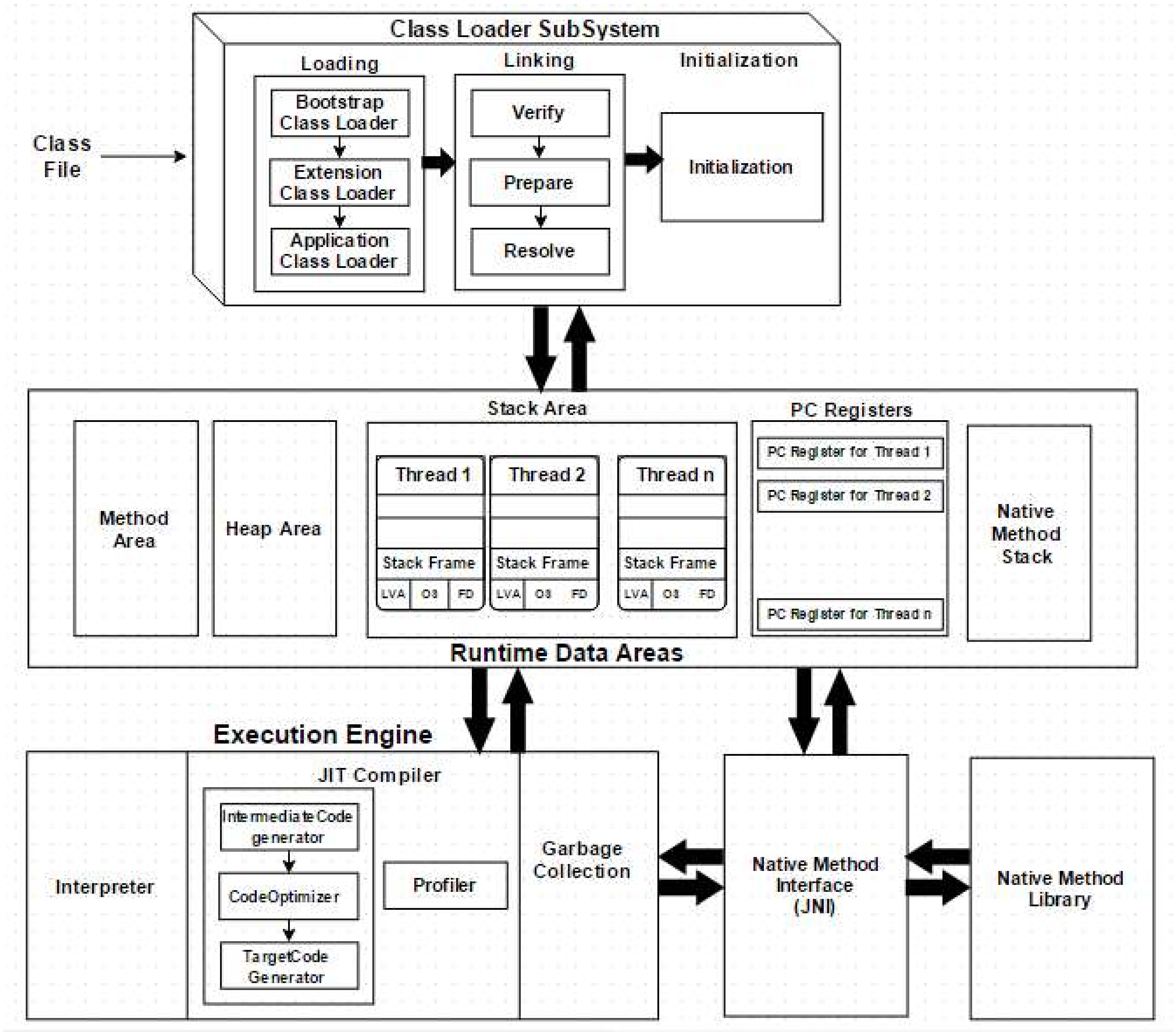
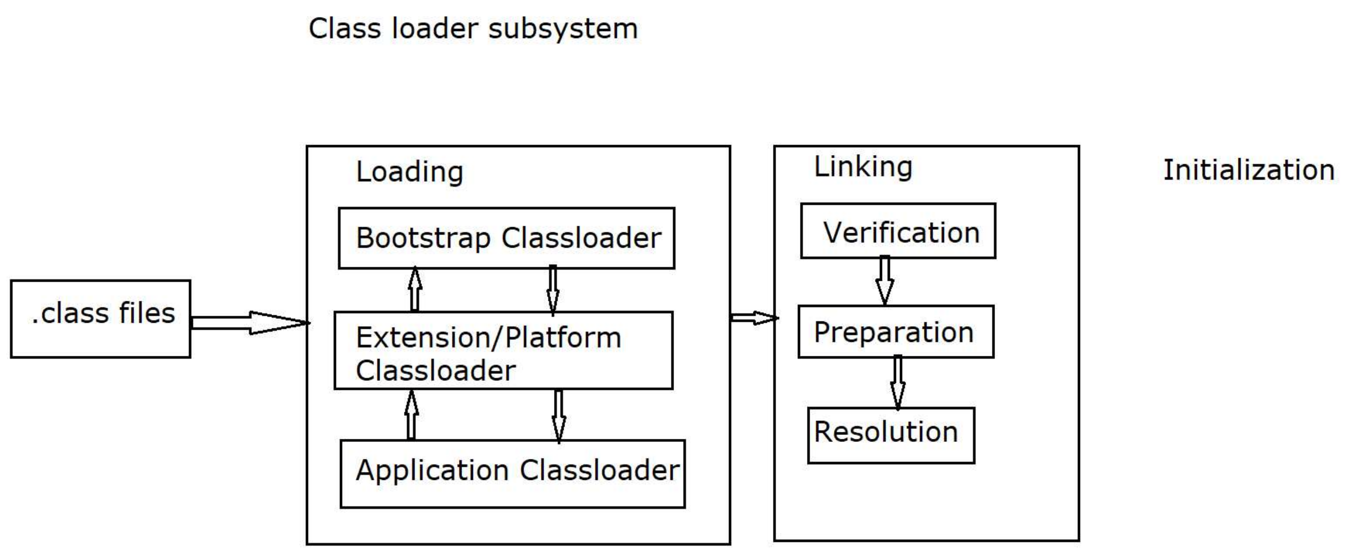
JVM Archititeture

@



JVM architecture (refer to a diagram) There are mainly three sub systems in the JVM

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



1. ClassLoader

This component is responsible for loading the class files to the method area (typcially in RAM) since JVM resides on the RAM and it performs : loading, linking, and initialization.

1.1 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.

1. The fully qualified name of the loaded class
2. variable information
3. immediate parent information
4. Type : whether it is a class or interface or enum

Note —Only for the first time, JVM creates an object from a class type object(java.lang.Class) for each loaded java class and stores that object in the heap.

1.1.2 The three main ClassLoaders in JVM,

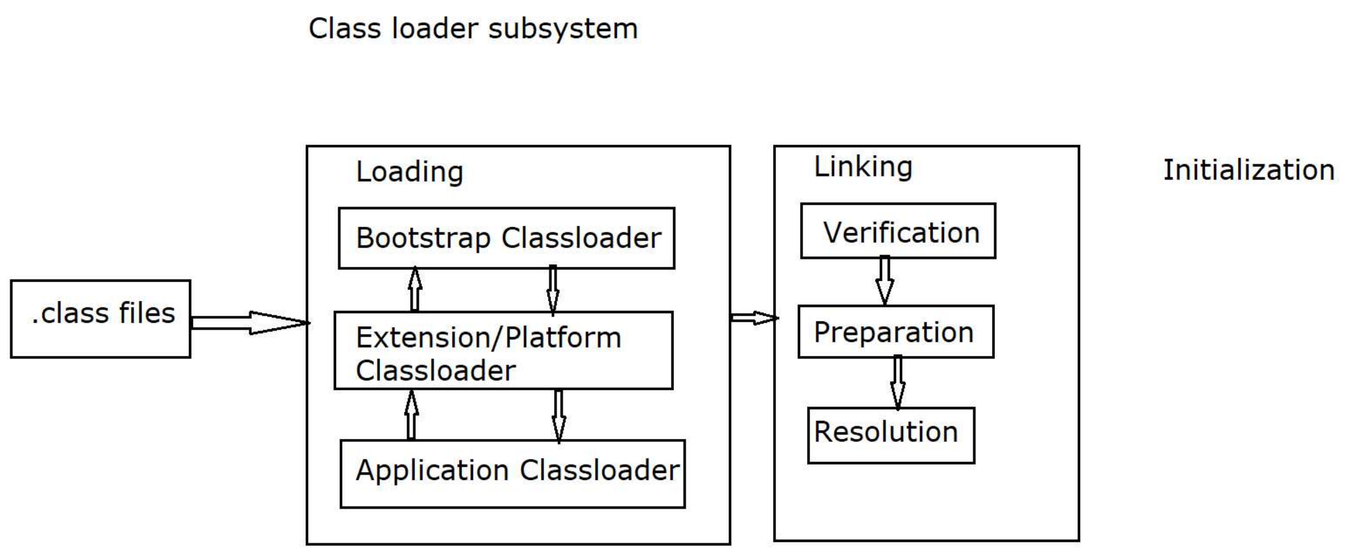
1. Bootstrap ClassLoader — This is the root class loader and it is the superclass of Extension/Platform ClassLoader. This loads the standard java packages which are inside the rt.jar/jrtfs.jar : java.base module (base file and some other core libraries.) eg : java.util.ArrayList

2. Extension/Platform ClassLoader — This is the subclass of the Bootstrap ClassLoader and a superclass of Applications ClassLoader. This is responsible for loading classes that are present inside the directory (jre/lib/ext). From JDK 9 onwards , it's replace by Platform class loader . Loads additional classes from other modules eg : java.sql

3. Application ClassLoader — This is the subclass of Extension/Platform ClassLoader and this is responsible for loading the class files from the classpath (classpath can be modified by adding the -classpath command-line option) eg : tester.TestMe

1.1.3 The four main principles in JVM,

1. **Visibility Principle** — This principle states that the ClassLoader of a child can see the class loaded by Parent, but a ClassLoader of parent can't find the class loaded by Child.
2. **Uniqueness Principle** — This principle states that a class loaded by the parent ClassLoader shouldn't be loaded by the child again. This ensures that there is no class duplicated.
3. **Delegation Hierarchy Principle** — This rule states that JVM follows a hierarchy of delegation to choose the class loader for each class loading request. Here, starting from the lowest child level, Application ClassLoader delegates the received class loading request to Extension ClassLoader, and then Extension ClassLoader delegates the request to Bootstrap ClassLoader. If the requested class is found in the Bootstrap path, the class is loaded. Otherwise, the request again transfers back to the Extension ClassLoader level to find the class from the Extension path or custom-specified path. If it also fails, the request comes back to Application ClassLoader to find the class from the System classpath and if Application ClassLoader also fails to load the requested class, then we get the run time exception — ClassNotFoundException.
4. **No Unloading Principle** — This states that a class cannot be unloaded by the Classloader even though it can load a class.



1.2 Linking

This process is divided into three main parts .

1.2.1 Verification

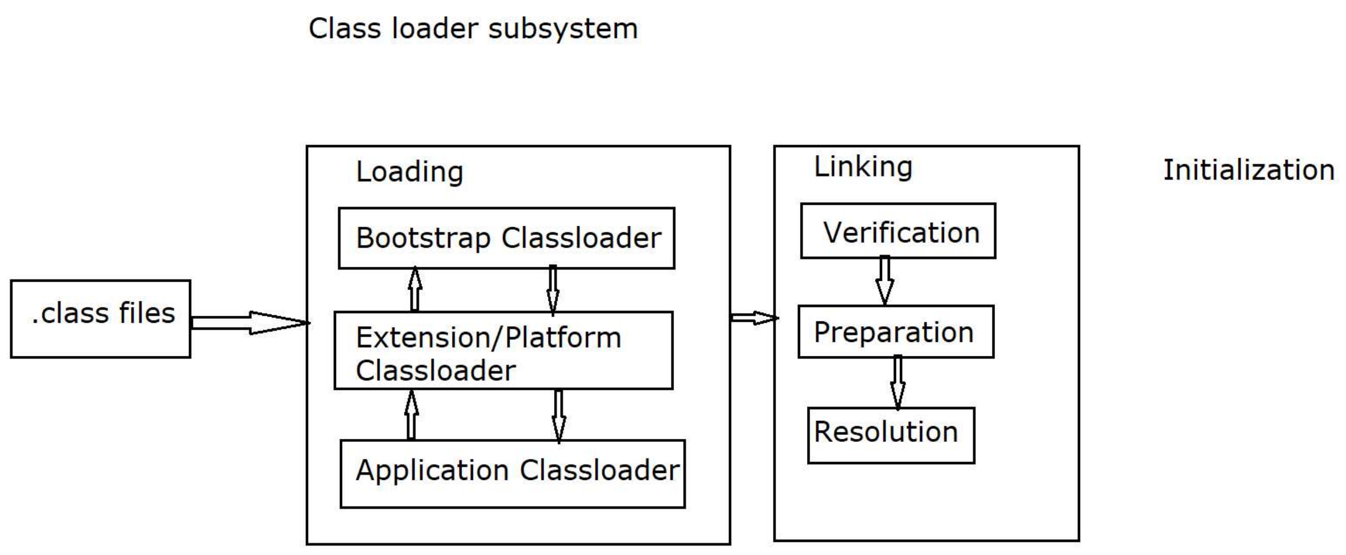
This phase check the correctness of the .class file. Byte code verifier will check the following: 2.1.1 whether it is coming from a valid compiler or not (Because anyone can create their own compiler).

2.1.2 whether the code has a correct structure and format.

if any of these are missing, JVM will throw a runtime exception called "java.lang.VerifyError" Exception. if not, then the preparation process will take place.

1.2.2 Preparation

In this phase, variables memory will be allocated for all static data members and assigned with default values based on the data types. eg : reference — null int — 0 boolean— false eg : static boolean active=true; So in this phase, it will check the code and the variable status in boolean type so JVM assigns false to that variable.



1.2.3 Initialization

In this phase, the original values will be assigned back to the static variables as mentioned in the code and a static initilizer block will be executed(if any). The execution takes place from top to bottom in a class and from parent to child in the class hierarchy.

Important : JVM has a rule saying that the initialization process must be done before a class becomes an active use.

Active use of a class are,

1. using new keyword. (eg: Vehicle car=new Vehicle();).
2. invoking a static method.
3. assigning value to a static field.
4. if a class is an initial class (class with main()method).
5. using a reflection API (Class's newInstance()method).
6. initializing a subclass from the current class.

There are four ways of initializing a class :

1. using new keyword — this will goes through the initialization process.

2. using clone(); method — this will get the information from the parent object (source object).

3. using reflection API (newInstance();) — this will goes through the initialization process.
4. using IO.ObjectInputStream(); — this will assign initial value from InputStream to all non-transient variable

2. Runtime Data Area

JVM memory is basically divided into five following parts,

2.1 Method Area

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). This is a shared resource.

2.2 Heap Area

This is where the information of all objects(state: non static data members) is stored and it’s a shared resource just like the method area

eg : Book book = new Book(...); So here, there is an instance of Book is created and it will be loaded into the Heap Area preceded by Book's class information loaded in the method area + creation of Class instance , in the heap.

Note — there is only one method area and one heap area per JVM.

2.3 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.

2.4 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.

2.5 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. Take a look at the following diagram,

eg scenario for Thread 1 (T1), M1(){ M2(); }

M2(){ M3(); } When the M1 method is called, the first frame will be created in the T1 thread and from there it will go to method M2 at that time the second frame will be created and from there it will go to method M3 as in the above demo code, so a new frame will be created under M2. Whenever the method exits, the stack frames will be destroyed respectively.

3. 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.

Mainly, Execution Engine has three main components for executing the Java classes,

Components of Execution Engine:-

3.1 Interpreter

This is responsible for converting bytecode into machine code. This is slow because of the line-by-line execution even though this interprets the bytecode quickly. The main disadvantage of Interpreter is that when the same method is called multiple times, every time a new interpretation is required and this will reduce the performance of the system. So this is the reason where the JIT compiler will run parallel to the Interpreter.

3.2 JIT Compiler (Just In Time Compiler)

This overcomes the disadvantage of the interpreter. The execution engine first uses the interpreter to execute the bytecode line-by-line and it will use the JIT compiler when it finds some repeated code. (Eg: calling the same method multiple times). At that time JIT compiler compiles the entire bytecode into native code (machine code). These native codes will be stored in the cache. So whenever the repeated method is called, this will provide the native code. Since the execution with the native code is quicker than interpreting the instruction, the performance will be improved.

3.3 Garbage Collector

This will check the heap area whether there are any unreferenced objects and it destroys those objects to reclaim the memory. So it makes space for new objects. This runs in the background and it makes the Java memory efficient.

There are two phases involved in this process, Mark — In this area, Garbage Collector identifies the unsued objects in the heap area. Sweep — In here, Garbage Collector removes the objects from the Mark. This process is done by JVM at regular intervals and it can also be triggered by calling System.gc() method.

3.4 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.

3.5 Native Method Libraries

These are the libraries that are written in other programming(non-java) languages such as C and C++ which are required by the Execution Engine. This can be accessed through the JNI and these library collections mostly in the form of .dll or .so file extension.

jvm native memory================> (focuse on heap memory) mark and sleep sleep = deletion

compaction=quicker faster memeory aloation