

## Class Loader Subsystem

It is mainly responsible for three activities.

* Loading
* Linking
* Initialization

**1. Loading** reads the .class file generate binary data and save it in method area.

There are **three** built-in classloaders in Java.

> **Bootstrap ClassLoader:** This is the super class loader for Extension class loader. It loads the rt.jar file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes

> **Extension ClassLoader:** This is the child class loader for Bootstrap and super class of system classloader. It loads the classes from the jar file inside JAVA\_HOME/jre/lib/ext directory.

> **System/Application ClassLoader:** This is the child class loader of extension class loader. It loads the class files from classpath. By default it is set to the current directory. You can change it using -cp or -classpath switch.

**2. Linking** performs verify, Prepare and Resolve.

> **Verify:** It ensures the correctness of a .class file, i.e. it checks whether this file is properly formatted and generated by a valid compiler.

If verification fails, we get a runtime exception java.lang.verifyError. This is done by the component BytecodeVerifier. Once this activity is completed then the class file is ready for compilation.

> **Preparation:** JVM allocates memory for class static variables and initializes the memory by default value.

> **Resolve:** Here we replace the symbolic references from the type with direct references. It is done by searching into the method area to locate the referenced entity.

**3. Initialization:** all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in the class hierarchy.

## **JVM Memory**

1. **Method area:** In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored including static variables.

\* From java 8, static variables are now stored in the Heap area.

2. **Heap area:** Information of all objects is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource.

3. **Stack area:** For Every thread, JVM creates one run-time stack which is stored here.

Every block of this stack is called activation record/stack frame which stores method calls.

All local variables of that method are stored in their corresponding frame.

After a thread terminates, its run-time stack is destroyed by JVM.

4. **PC Registers:** Store address of current execution instruction of the thread. Each thread has a separate PC Register.

5. **Native Method Stack:** For every thread, a separate native stack is created. It stores native method information.

**Execution Engine**

Execution engine executes the ‘.class’ (bytecode).

\* Resolution:

Resolution is the process of mapping symbolic references to direct references during the class loading phase in Java. It's a crucial step that allows Java to link classes, methods, and fields together correctly for runtime execution. This phase ensures that when your Java program runs, it can find the appropriate classes, methods, and fields it needs, even if they are defined in different classes or libraries.

Here's a more detailed breakdown of what happens during the resolution phase:

* Symbolic References: In Java source code, references to classes, methods, and fields are often symbolic. This means they are represented by names or descriptors rather than direct memory addresses. For example, when you call a method or access a field in your code, you typically use its name, and the Java compiler represents this reference symbolically.
* Method Area: The JVM maintains a memory area called the "method area" (or "constant pool" in older versions of the JVM). This area stores information about classes, methods, fields, and their symbolic references. During class loading, information about classes is loaded into the method area, and symbolic references within the classes' bytecode are also stored there.
* Resolution Process: When a class is being loaded or initialized, and it contains symbolic references, the JVM performs a process known as resolution:  
  a. Class Resolution: If a class contains references to other classes (e.g., when it extends another class or implements an interface), the JVM resolves these references by looking up the fully qualified names of the classes and ensuring they are available in the classpath. If necessary, the referenced classes are loaded and initialized.  
  b. Method Resolution: For symbolic method references (e.g., method calls), the JVM searches the method area for the class where the referenced method is declared. It then verifies the method's existence, accessibility, and its signature. If found, the symbolic reference is replaced with a direct reference to the actual method in memory.  
  c. Field Resolution: Similar to method resolution, symbolic field references (e.g., accessing a field) are resolved by locating the field's declaration and verifying its existence, accessibility, and type. A direct reference to the field is established in memory.
* Direct References: Once the resolution phase is complete, all symbolic references within the class have been replaced with direct references to the actual classes, methods, and fields in memory. This means that when your Java program runs, it can directly access and execute the methods and fields without having to perform expensive symbolic lookups at runtime.

In summary, the resolution phase ensures that Java programs can efficiently and correctly link classes, methods, and fields during runtime. It optimizes performance by converting symbolic references into direct references, allowing for faster and more predictable execution.