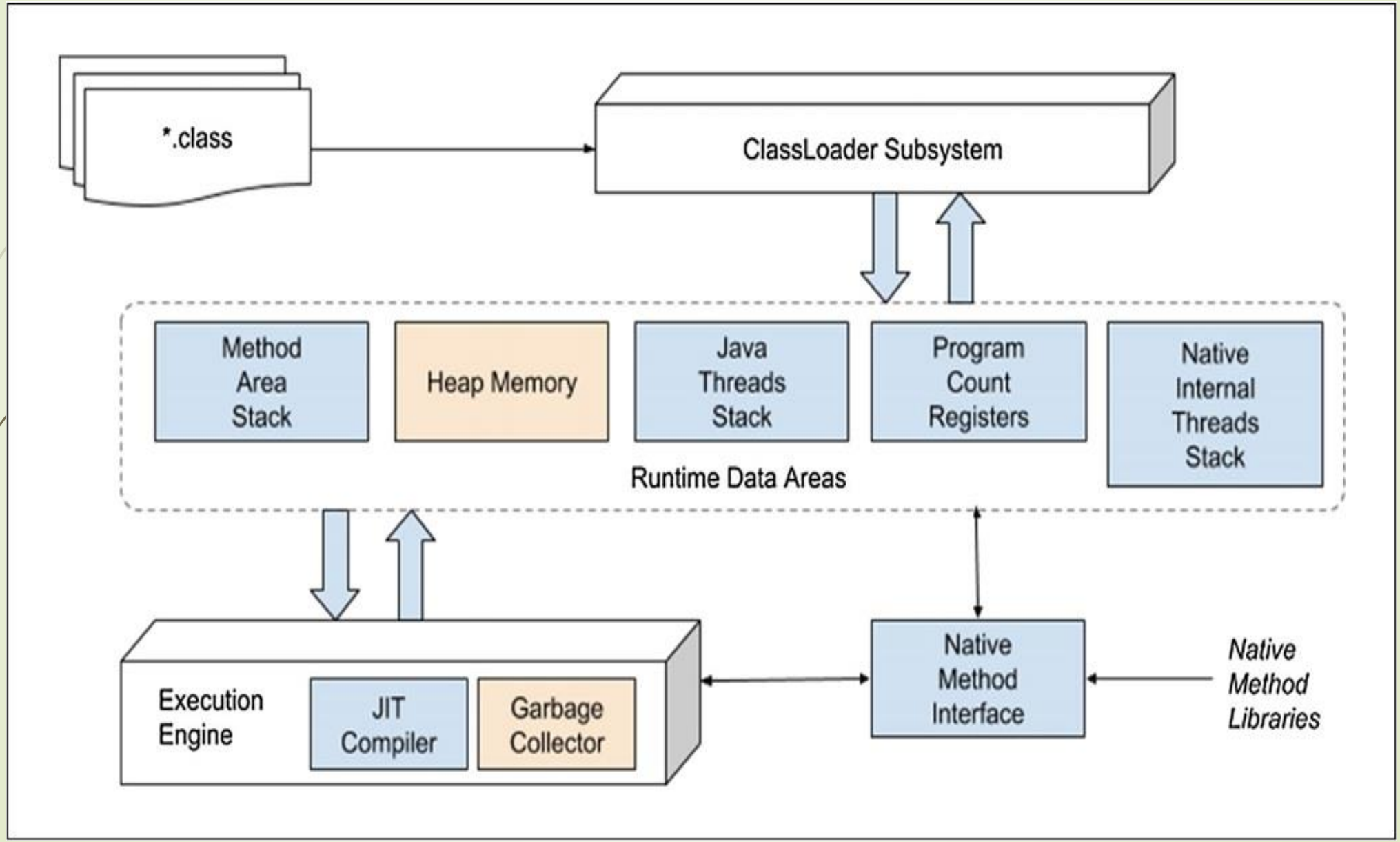
JVM Overview

JVM Architecture

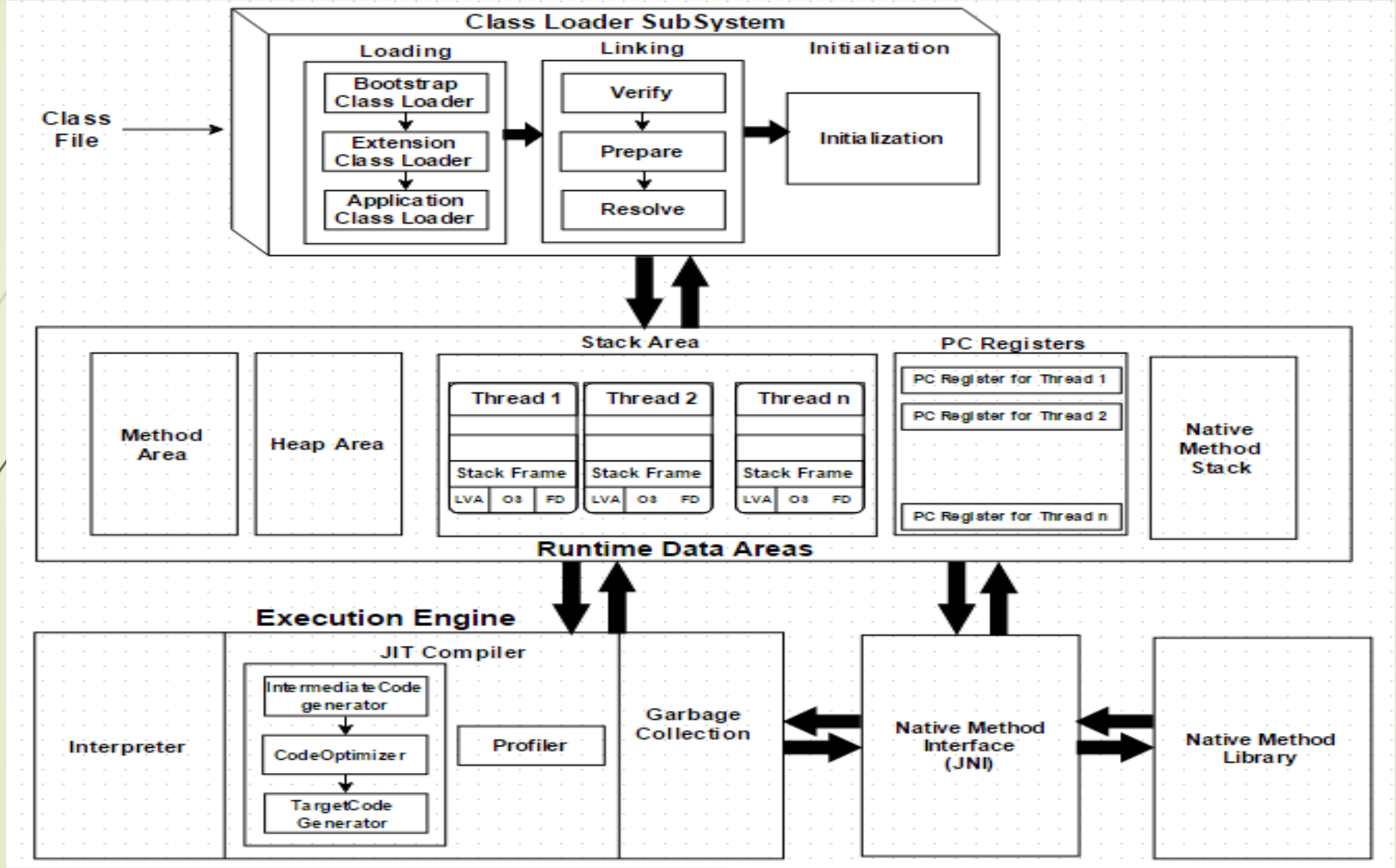


JVM Architecture

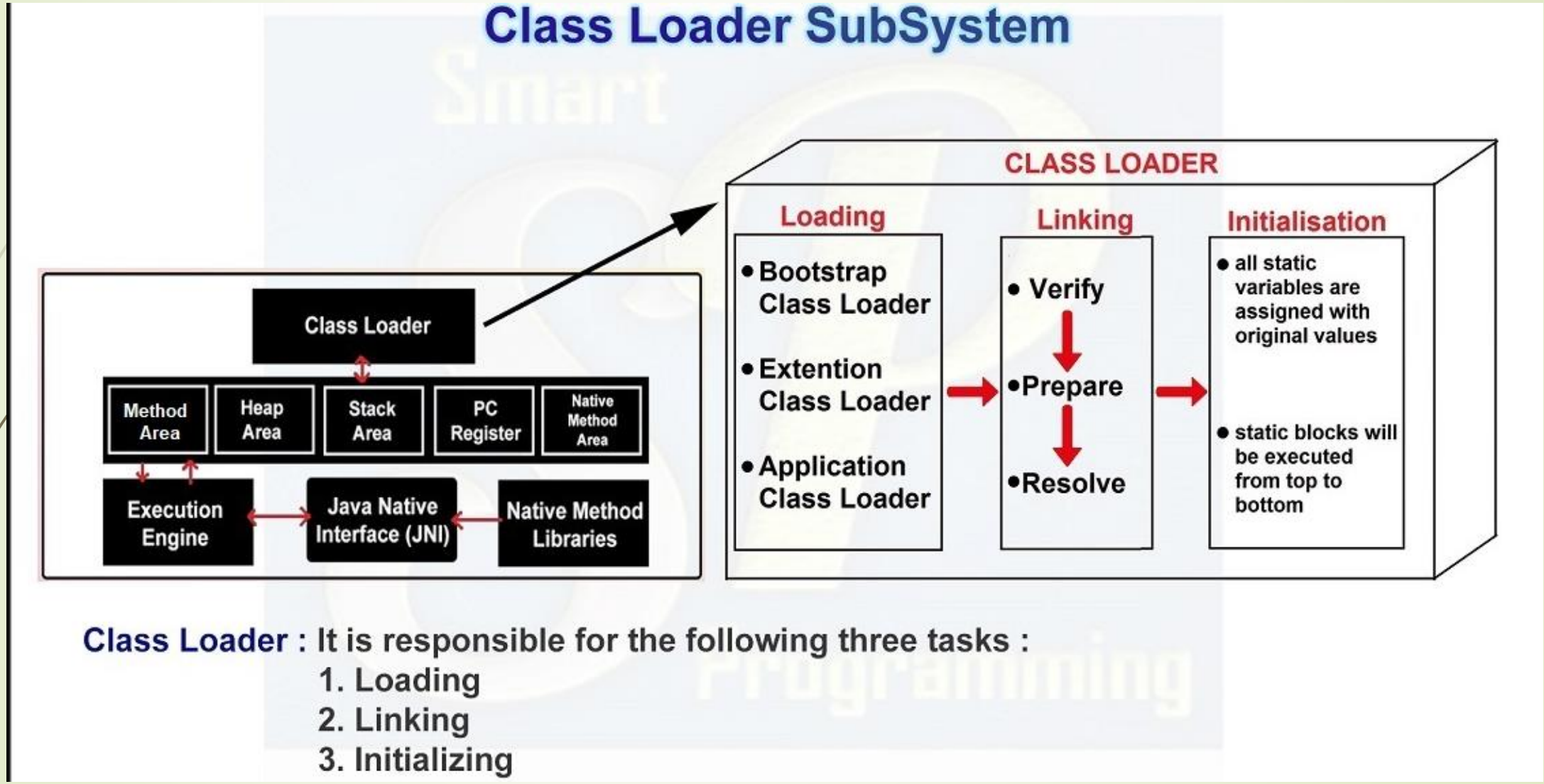
- JVM architecture -There are mainly three sub systems in the JVM

1. Class Loaders
2. Runtime Memory/Data Areas
3. Execution Engine

JVM Architecture detailed



Class Loaders



Class Loader : It is responsible for the following three tasks :

1. Loading
2. Linking
3. Initializing

Class Loaders

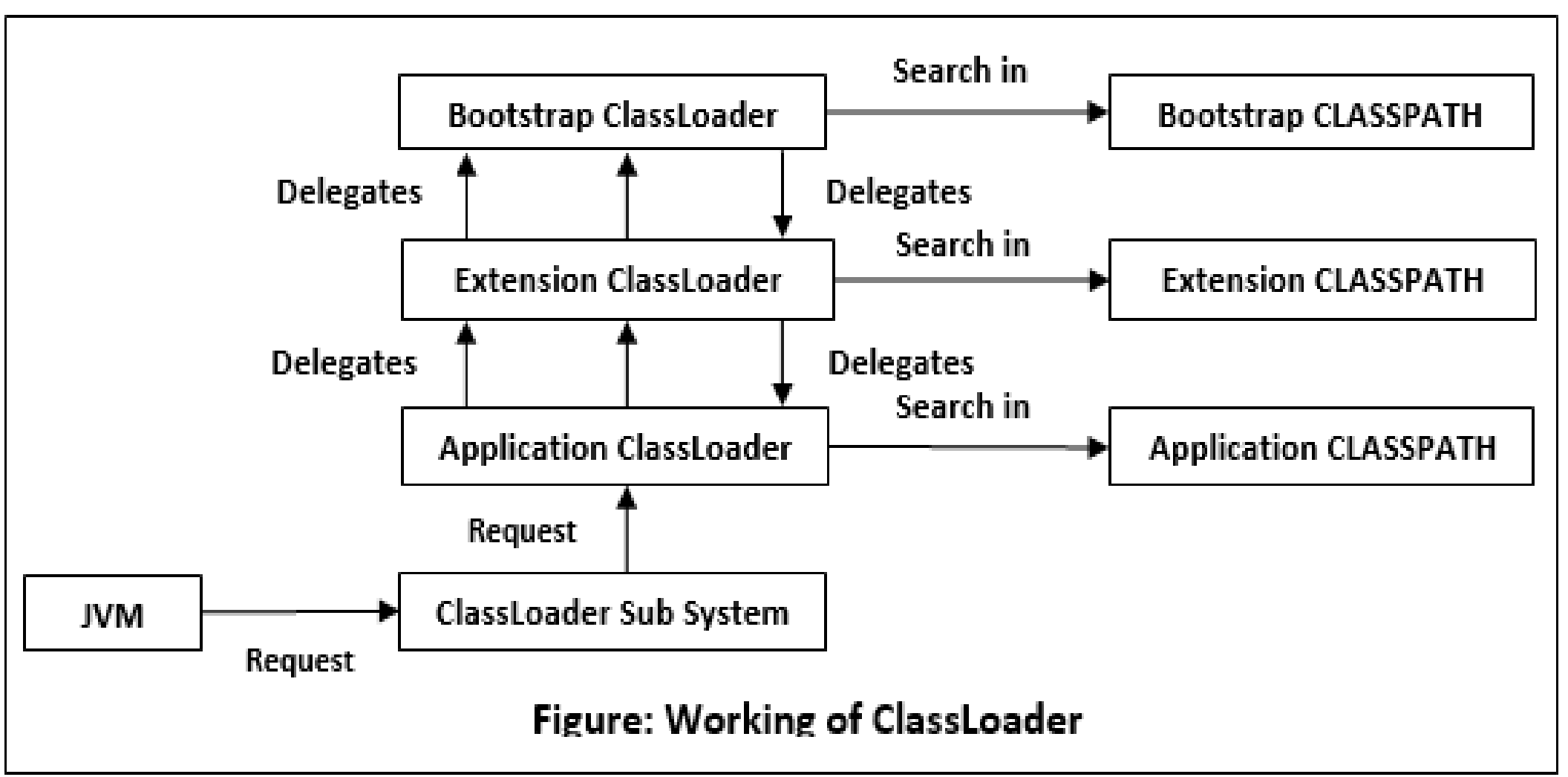


Figure: Working of ClassLoader

Class Loaders

- Class loaders are responsible for loading the class files to the method area
- Class loaders perform 3 steps : **Loading, Linking, and Initialization.**
  - **Loading**
    - This process usually starts with loading the main class (class with the main()method).
    - Class Loader 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
    - For 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.

Class Loaders types

- **The three main ClassLoaders in Java,**
  - **Bootstrap Class Loader** — Its root class loader and it is the superclass of Extension Class Loader. This loads the standard java packages which are inside the **rt.jar/jrtfs.jar** : **java.base module** eg : **java.lang.System**
  - **Platform/Extension Class Loader** — its a subclass of the Bootstrap Class Loader and a superclass of Applications Class Loader. This is responsible for loading classes that are present inside the directory (jre/lib/ext). eg : **java.sql**
  - **Application Class Loader** — it's a subclass of Extension Class Loader and it is responsible for loading the class files from the classpath (class path is specified b user) eg : Com.acts.Employee

Class Loaders Principles

- There are four main principles in JVM,
- **Visibility Principle** — ClassLoader of a child can see the class loaded by Parent, but a ClassLoader of parent can’t find the class loaded by Child.
- **Uniqueness Principle** —a class loaded by the parent Class Loader shouldn’t be loaded by the child again. It avoid duplication.
- **No Unloading Principle** — A class cannot be unloaded by the Classloader even though it can load a class.

Class Loaders Principles

- **Delegation Hierarchy Principle** —JVM follows a hierarchy of delegation to choose the class loader for each class loading request.
  - Starting from the lowest child level, Application Class Loader delegates the received class loading request to Extension Class Loader, and then Extension Class Loader delegates the request to Bootstrap Class Loader.
  - 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.**

Linking

- **Linking** : This process is divided into three main parts .

1. Verification : It checks correctness of the .class file.

Byte code verifier will check:

1.1 If it is coming from a valid compiler or not.

1.2 If 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.

Linking

2. **Preparation** :
   - 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, static boolean active=true;
   - it will check the code and the variable status in boolean type so JVM assigns false to that variable.
3. **Resolution**
   - 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.

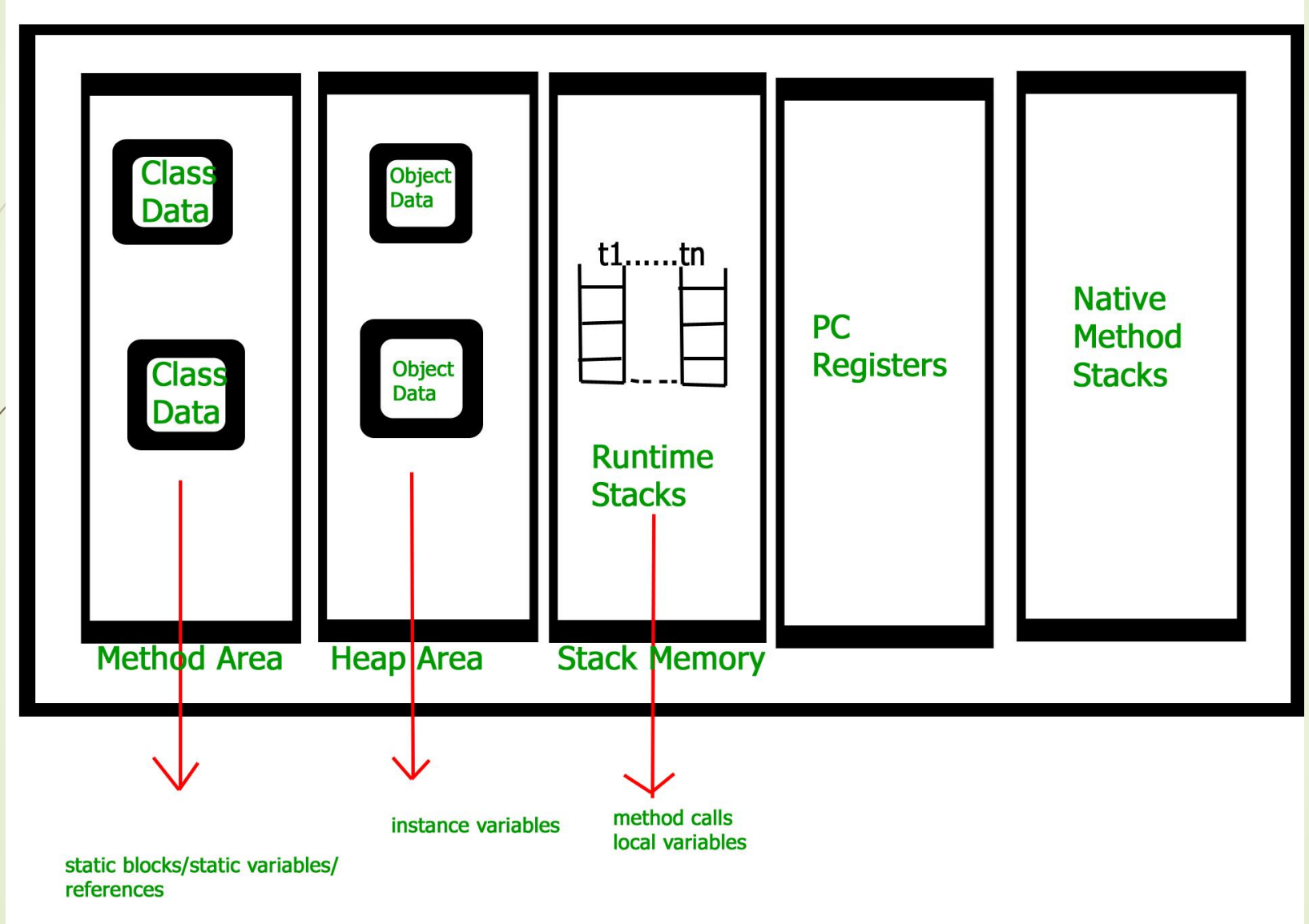
Initialization

- **Initialization** : The original values will be assigned back to the static variables as mentioned in the code and a static initializer 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.
- Initialization process must be done before a class becomes an active use.
- Active use of a class are,
  - using new keyword or instantaion.
  - invoking a static method. Vehicle
  - assigning value to a static field.
  - if a class is an initial class (class with main()method).
  - using a reflection API (Class's newInstance()method).
  - initializing a subclass from the current class.

Initialization

- There are four ways of initializing a class :
- using new keyword — this will goes through the initialization process.
- using clone(); method — this will get the information from the parent object (source object).
- using reflection API (newInstance();) — this will goes through the initialization process.
- using IO.ObjectInputStream(); — this will assign initial value from InputStream to all non-transient variable

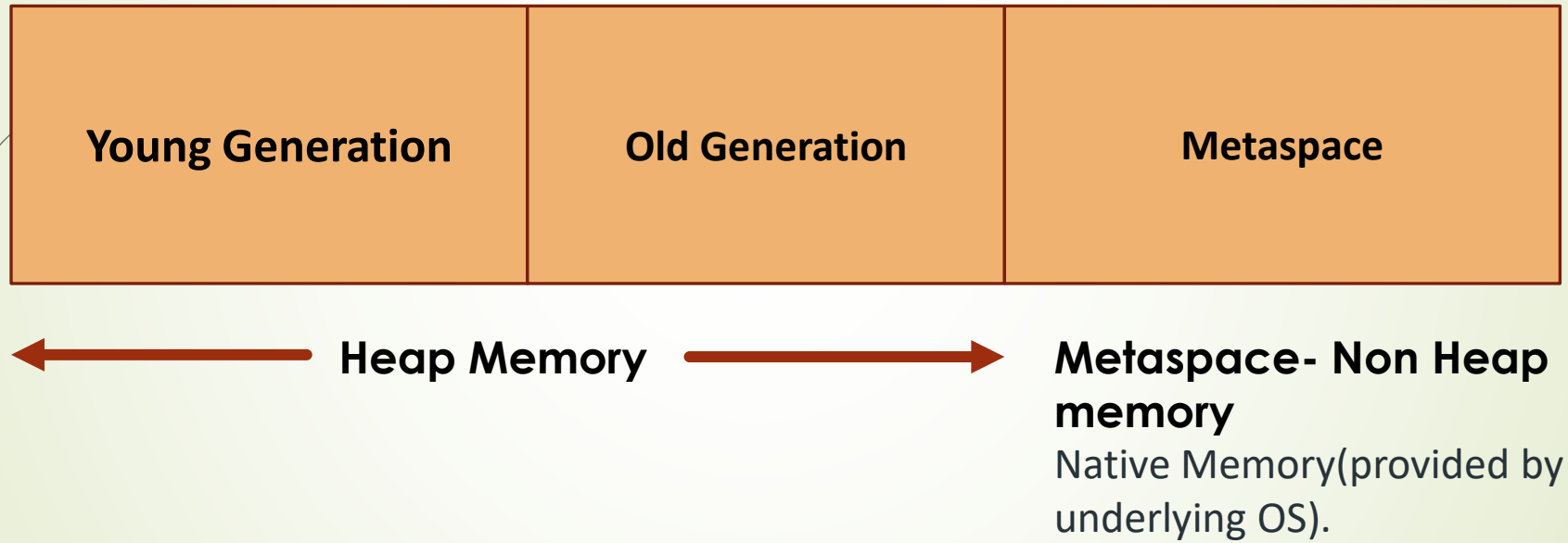
JVM Memory


Class Data
Class Data
Object Data
Object Data
t1.....tn
Runtime Stacks
PC Registers
Native Method Stacks
Method Area
Heap Area
Stack Memory
static blocks/static variables/ references
instance variables
method calls
local variables

JVM Memory

- JVM memory is divided into five following parts,
- **1. Method Area** : 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).
- **2. Heap Area :** The information of all objects(state: non static data members) is stored and it's a shared resource just like the method area eg : Employee emp = new Employee(...);
  - There an instance of Employee is created and it will be loaded into the Heap Area preceded by Employee 's class information loaded in the method area + creation of Class< Employee instance , in the heap.
- **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.
- **4. PC Register (Program Counter Register) :** It 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.
- **5. Native Method Area :** It 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.

JVM Memory area



Heap area

- The heap area is one of the most important memory areas of JVM. Here, all the java objects are stored. The heap is created when the JVM starts. The heap is generally divided into two parts. That is:

1. **Young Generation(Nursery):** All the new objects are allocated in this memory. Whenever this memory gets filled, the garbage collection is performed. This is called as ***Minor Garbage Collection***.

1. **Old Generation:** All the long lived objects which have survived many rounds of minor garbage collection is stored in this area. Whenever this memory gets filled, the garbage collection is performed. This is called as ***Major Garbage Collection*.**

Metaspace

- **Metaspace** by default auto increases its size depending on the underlying OS. It avoid **OutOfMemoryError**
- It uses Native Memory(provided by underlying OS). Comparatively efficient Garbage collection. Deallocate class data concurrently and not during GC pause.
- A new flag is available (**MaxMetaspaceSize**), allowing you to limit the amount of native memory used for class metadata
- **Metaspace garbage collection**

- Garbage collection of the dead classes and classloaders is triggered once the class metadata usage reaches the “MaxMetaspaceSize”.
- Proper monitoring & tuning of the Metaspace will obviously be required in order to limit the frequency or delay of such garbage collections.

Execution Engine

- **Execution Engine :** Here 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.
- Components of Execution Engine : **Interpreter, JIT Compiler and Garbage Collector**
- **Interpreter**
  - It converts bytecode into machine code.
  - Its 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.

Execution Engine

- **JIT Compiler (Just In Time Compiler) : It** 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.
  - 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.

Execution Engine

- **Garbage Collector :** It checks 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.
- Gc runs in background and it makes the Java memory efficient.
- There are two phases involved in this process,
  - Mark —Garbage Collector identifies the unused objects in the heap area.
  - Sweep —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.

Java Native Library

- **Java Native Interface (JNI)**
  - Used to interact/load the Native(non-java) Method libraries (C/C++) required for the execution.
  - Its allow JVM to call those libraries to overcome the performance constraints and memory management in Java.

- **Native Method Libraries**
  - Libraries that are written in other programming 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.