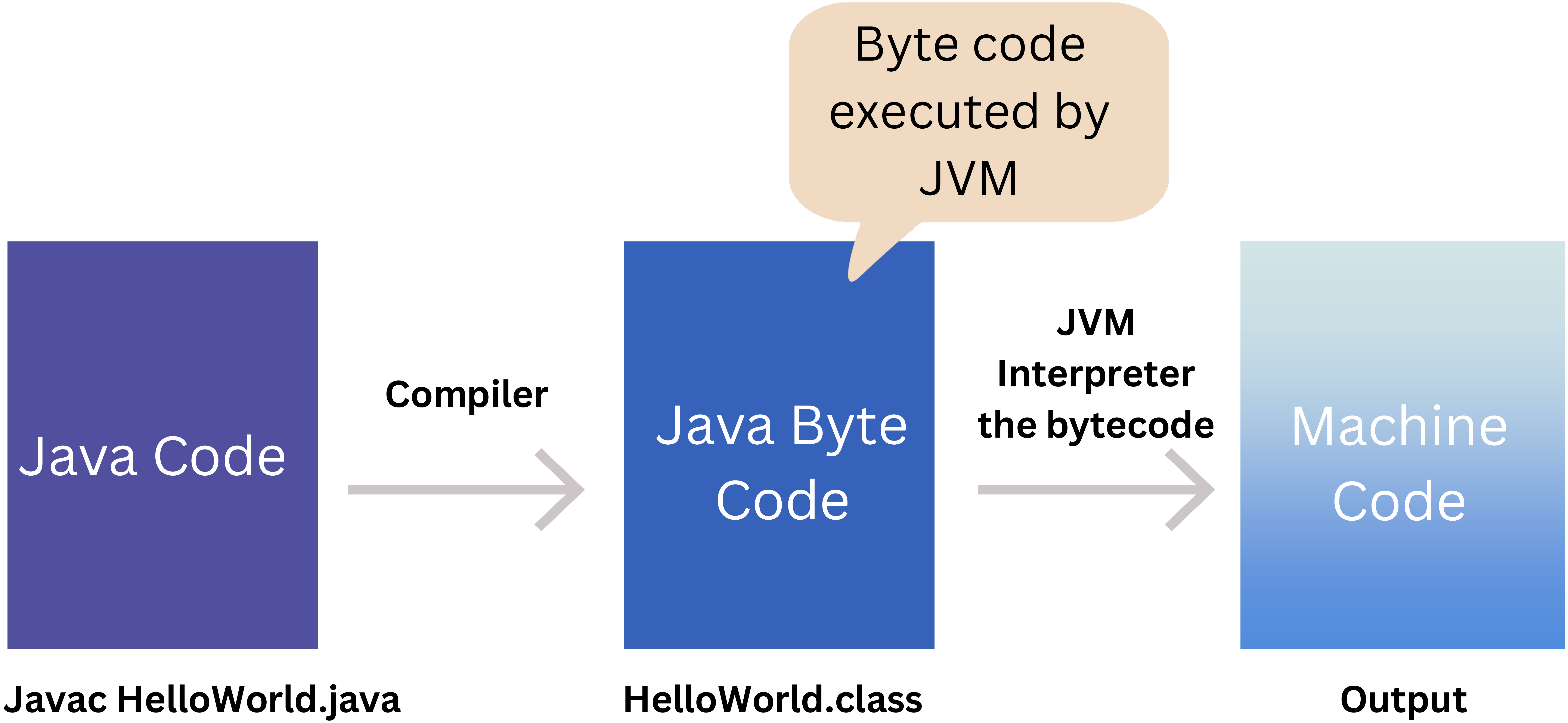
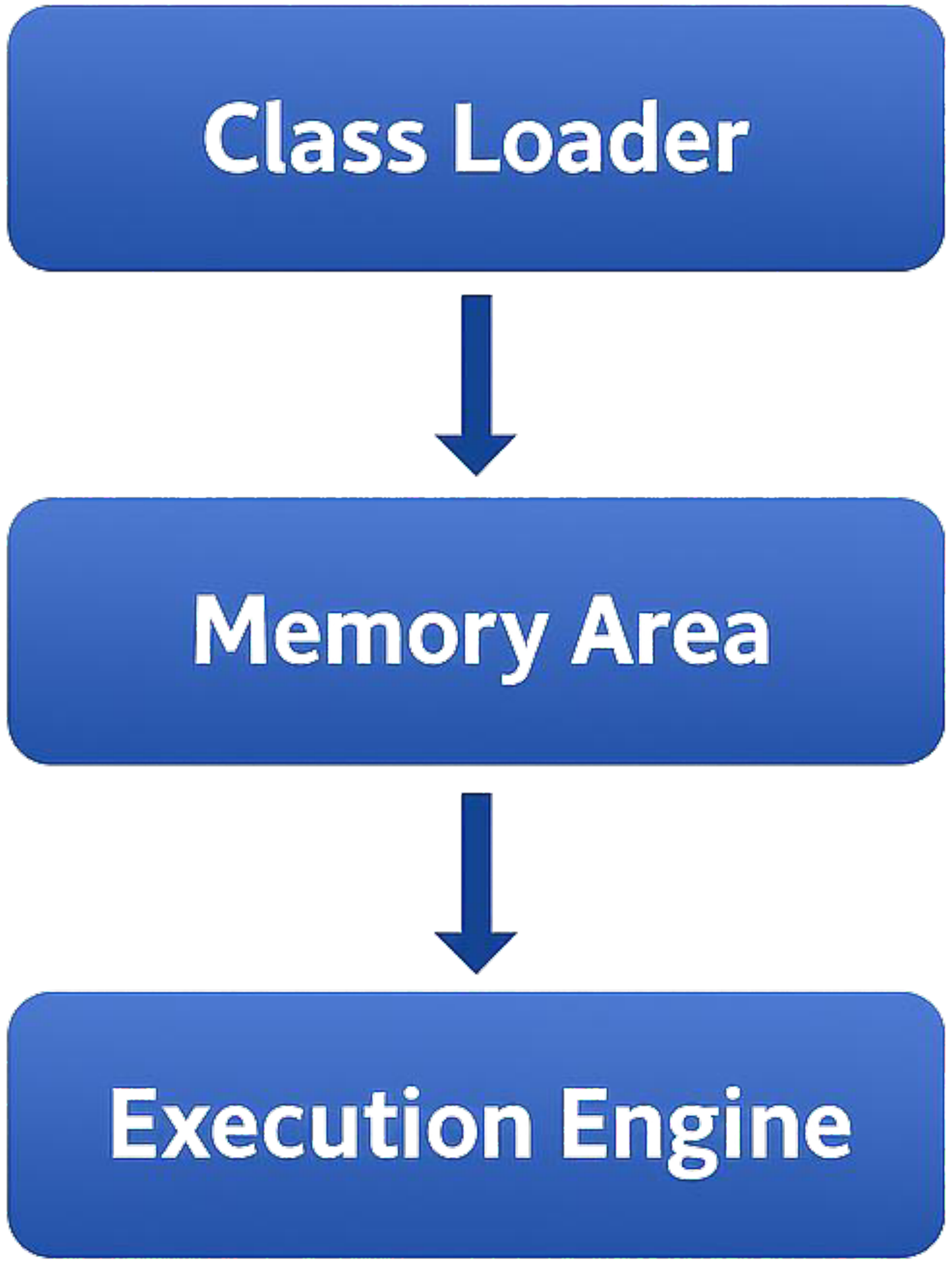
JVM Architecture



JVM

JVM mainely divided into three parts:



JVM

Class Loader Subsystem

Loading

- Bootstrap class Loader
- Extension Class Loader
- Application class Loader

Linking

- Verify
- Prepare
- Resolve

Initialization

- All static variables are assign with value
- Static block will be executed from top to bottom

Memory Area

Execution Engine

JVM-Class Loader

Loading

Bootstrap class loader

↓

Extension class loader

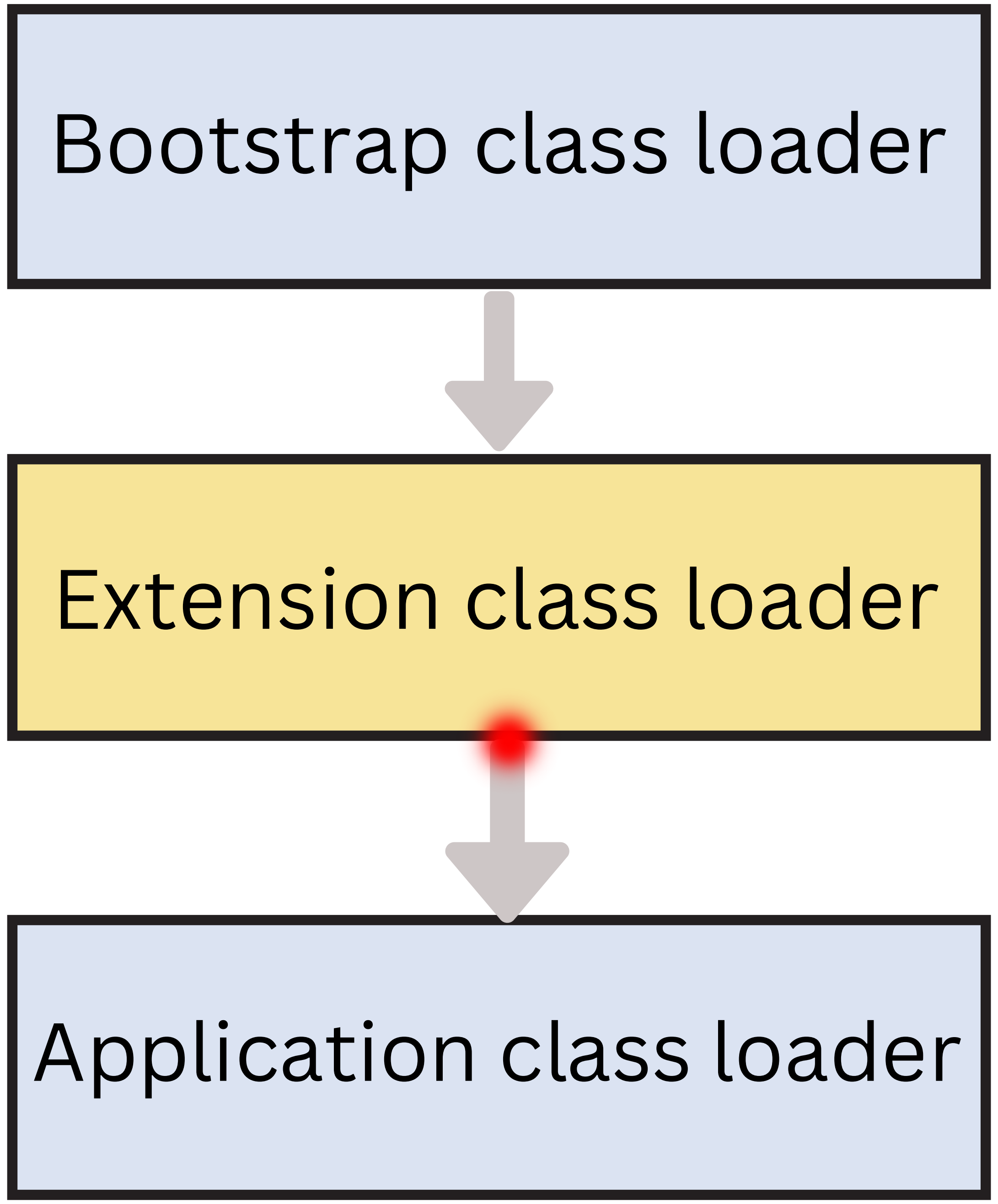
↓

Application class loader

It is the first class loader that is responsible for loading the core Java libraries located in the ‘<JAVA_HOME>/jre/lib/*.jar’.

JVM-Class Loader

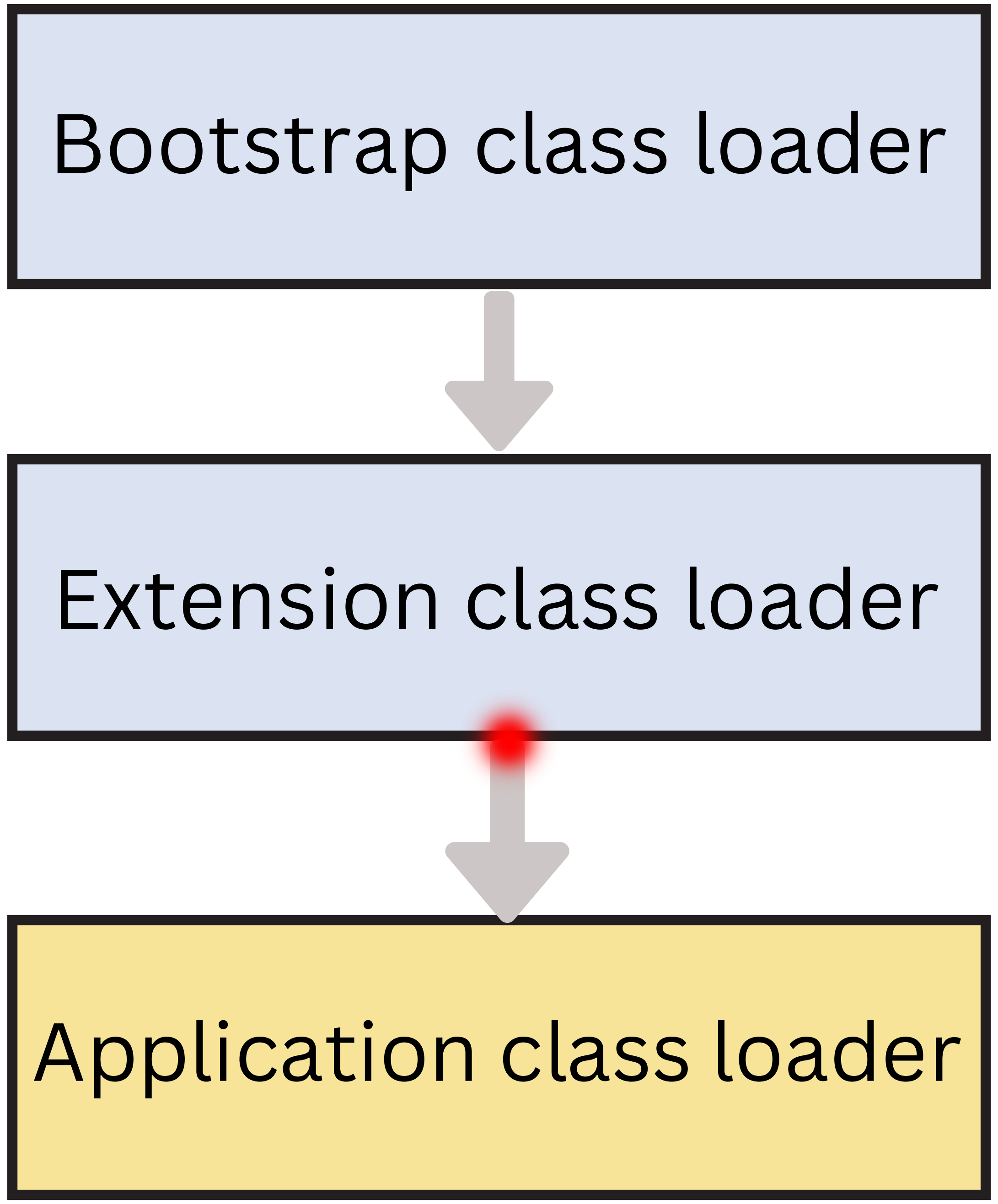
Loading



- **It is a child class of Bootstrap class loader.**
- **It is responsible of loading all classes from the extension class path in Java.**
- **Extension class path is: '<JAVA_HOME>/jre/lib/ext' .**

JVM-Class Loader

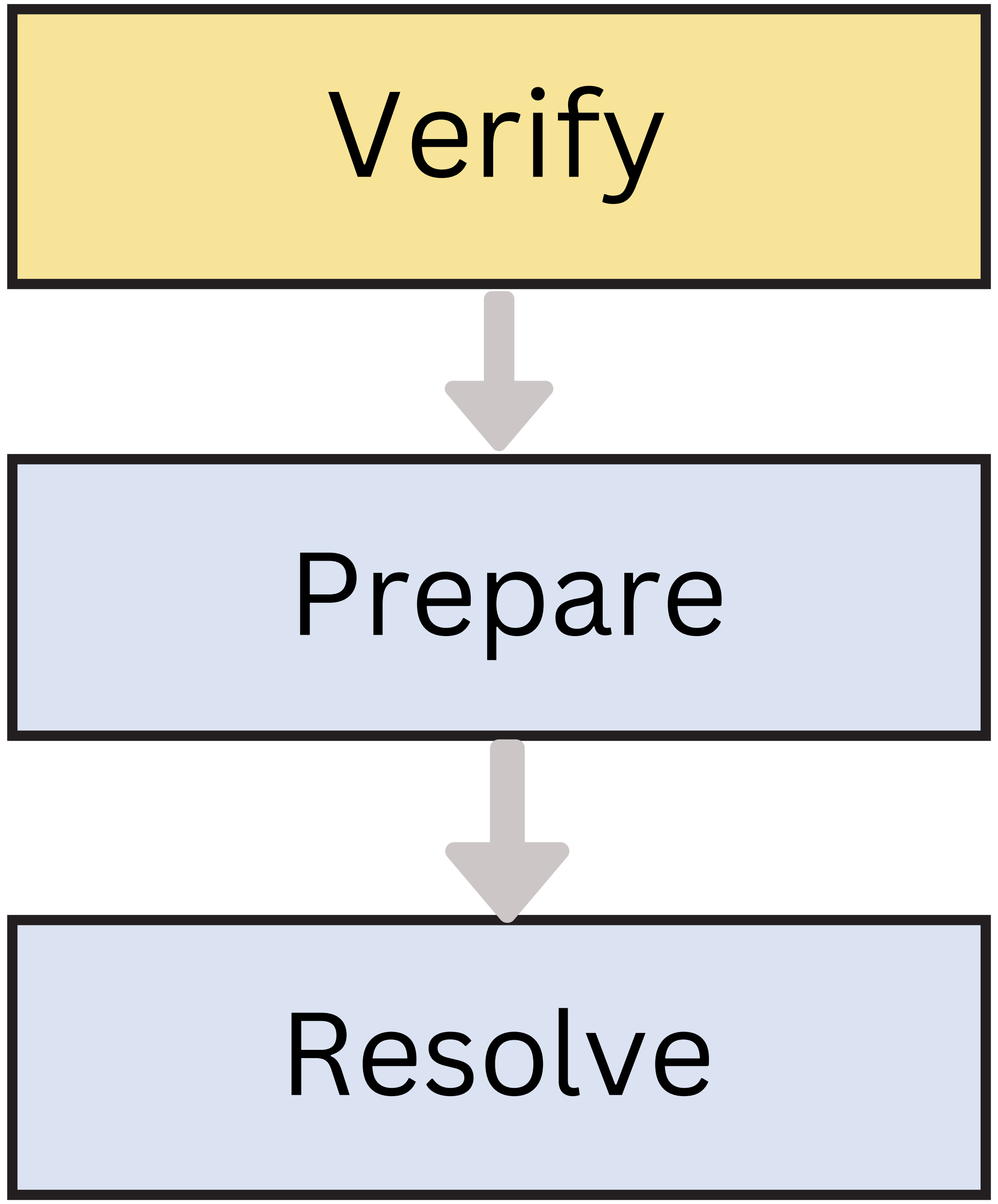
Loading



- **It is a child class of Extension class loader.**
- **It is responsible of loading classes from the application classpath.**
- **Application class loader dealing with application-specific classes.**

JVM-Class Loader

Linking

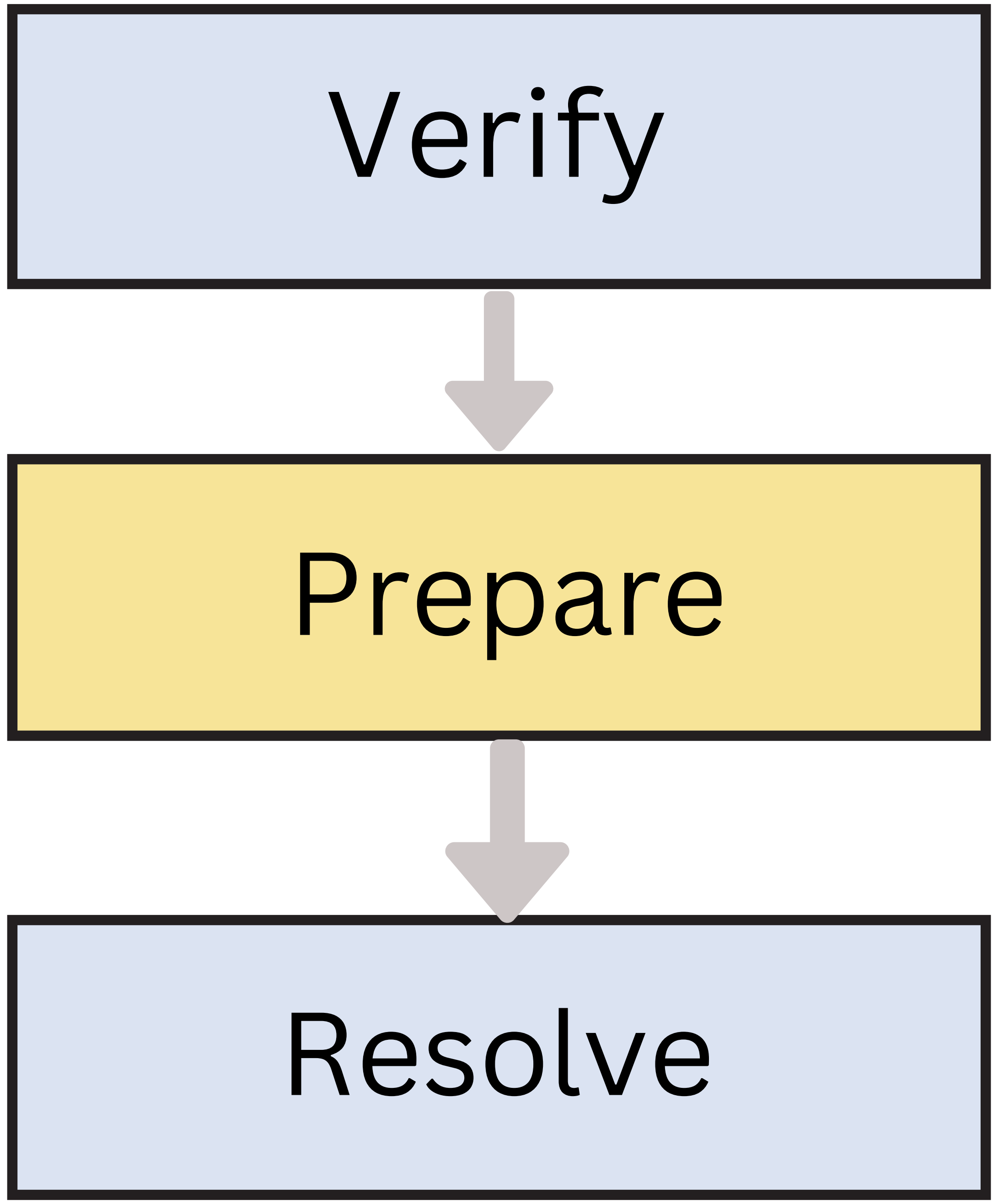


Verification:

- **It checks whether the ".class" file is generated by valid compiler or not.**
- **If verification fails, it throws java.lang.VerifyError.**

JVM-Class Loader

Linking



Preparation:

- JVM allocates memory for class variables and initializes them with default values.

JVM-Class Loader

Linking

Verify

↓

Prepare

↓

Resolve

Resolution:

It is the process of replacing the **symbolic reference** with direct reference to actual memory addresses.

JVM-Class Loader

Linking

Verify

↓

Prepare

↓

Resolve

For Example:

```
public class Main {
    public static void main(String[] args) {
        // Helper helper = new Helper();
        helper.performTask();
    }
}
```

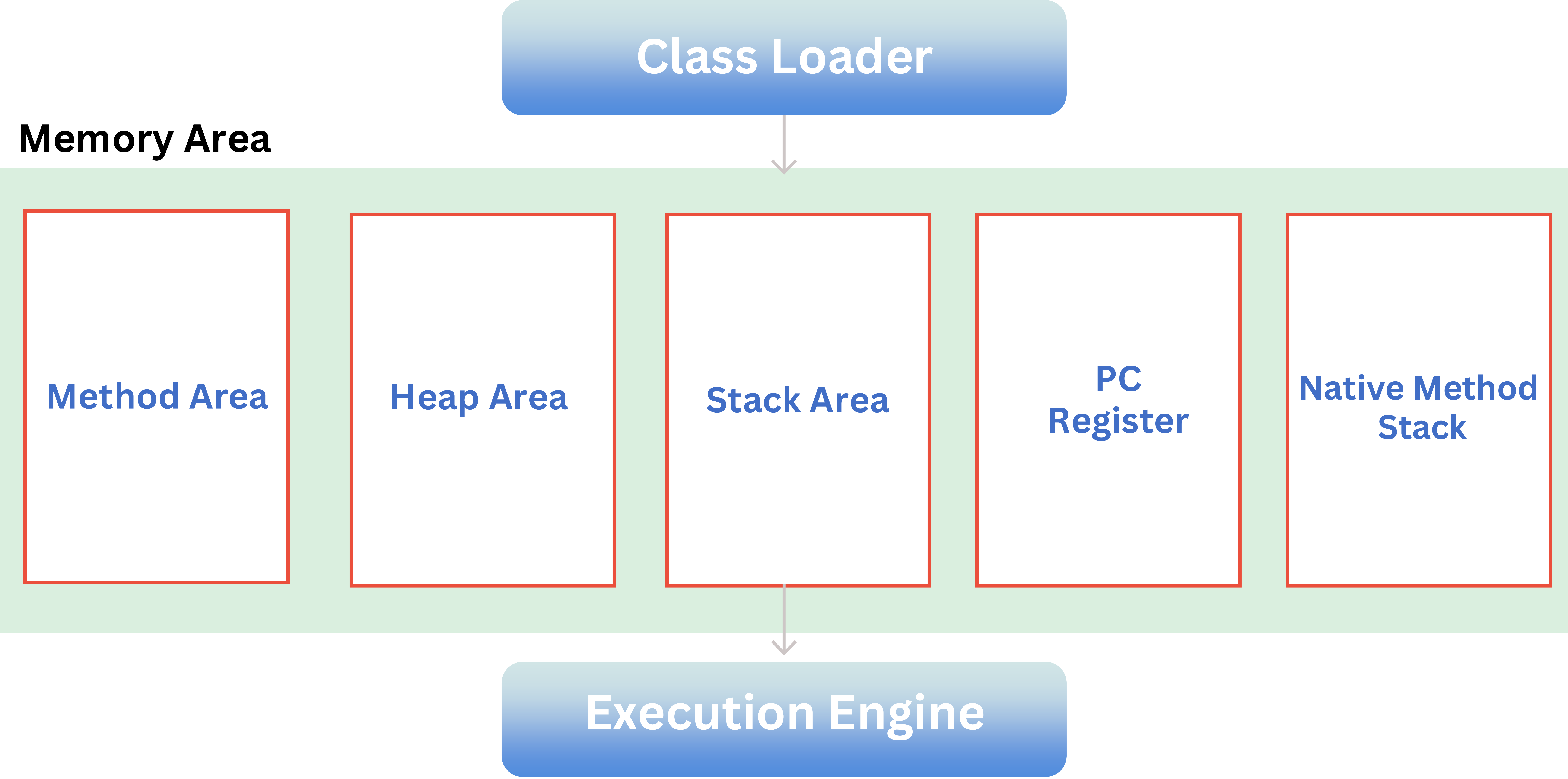
- The reference to **Helper** and its method **performTask** in the **Main** class is **symbolic**. The actual memory locations are not known at compile time.
- During the **linking** phase, the **JVM loads the Main class**. At this point, symbolic references to **Helper** and its method need to be resolved.
- The **resolution** step finds the actual memory addresses for the **Helper** class and its **performTask** method. It ensures that the **Main** class can call the correct method on the correct object.

JVM-Class Loader

Initialization

- All static variables are assigned with value.
- Static block will be executed from top to bottom.

JVM-Memory Area



JVM-Class Loader

Method Area

- Method Area
- Heap Area
- Stack Area
- PC Register
- Native Method Stack

- It **stores class level information** like class name, method and variable information.
- Stores static variables.
- The method area is shared among all threads running in the JVM. (Not thread safe)
- Only one Method area per JVM.

JVM-Class Loader

Method Area

- Method Area
- Heap Area
- Stack Area
- PC Register
- Native Method Stack

Method Area, also known as the **"Permanent Generation"** in older JVM versions (up to Java 7) or as the "Metaspace" in Java 8 and later.

- It **stores class level information** like class name, method and variable information.
- Stores static variables.
- The method area is shared among all threads running in the JVM. (Not thread safe)
- Only one Method area per JVM.

JVM-Class Loader

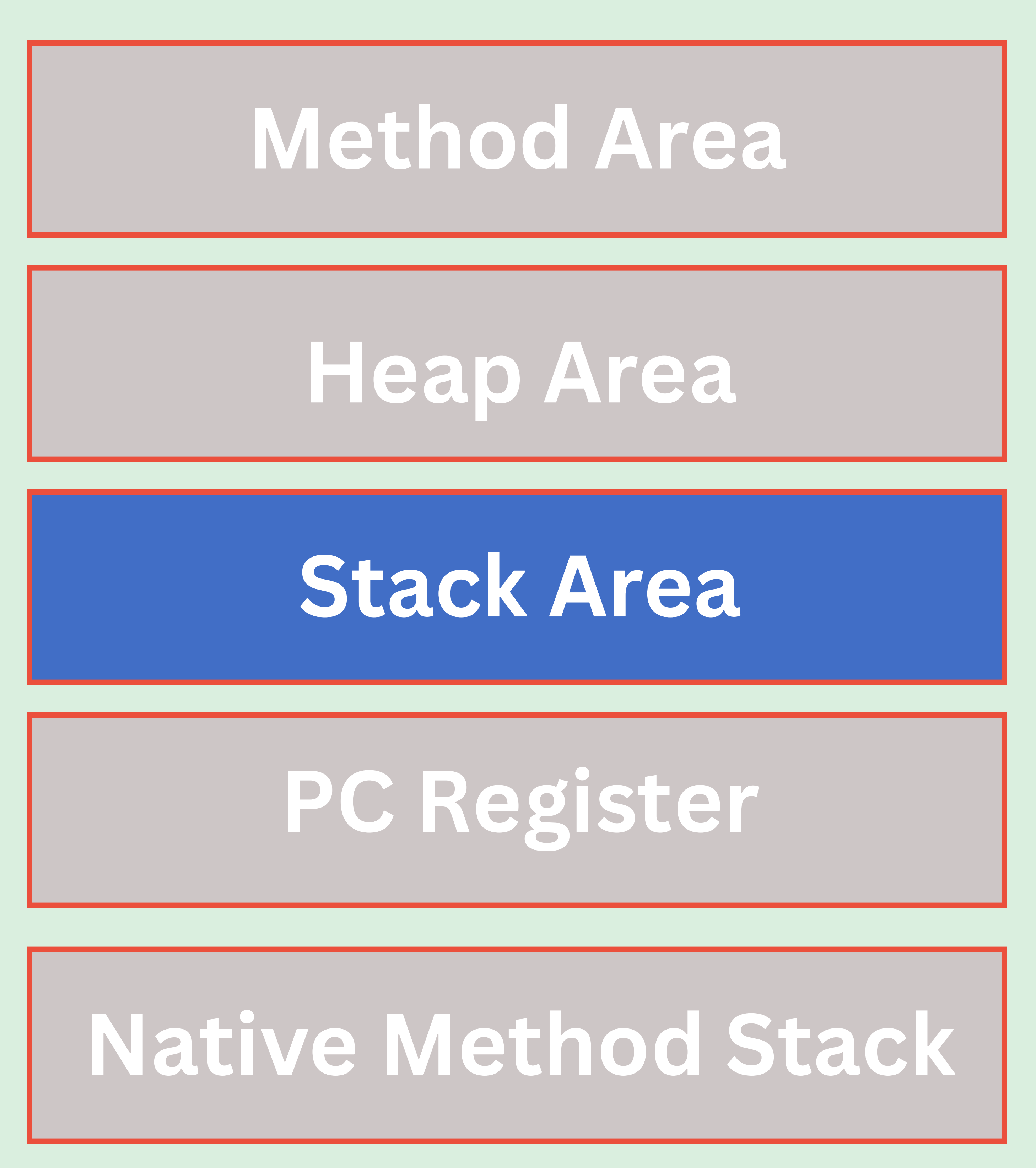
Heap Area

- Method Area
- **Heap Area**
- Stack Area
- PC Register
- Native Method Stack

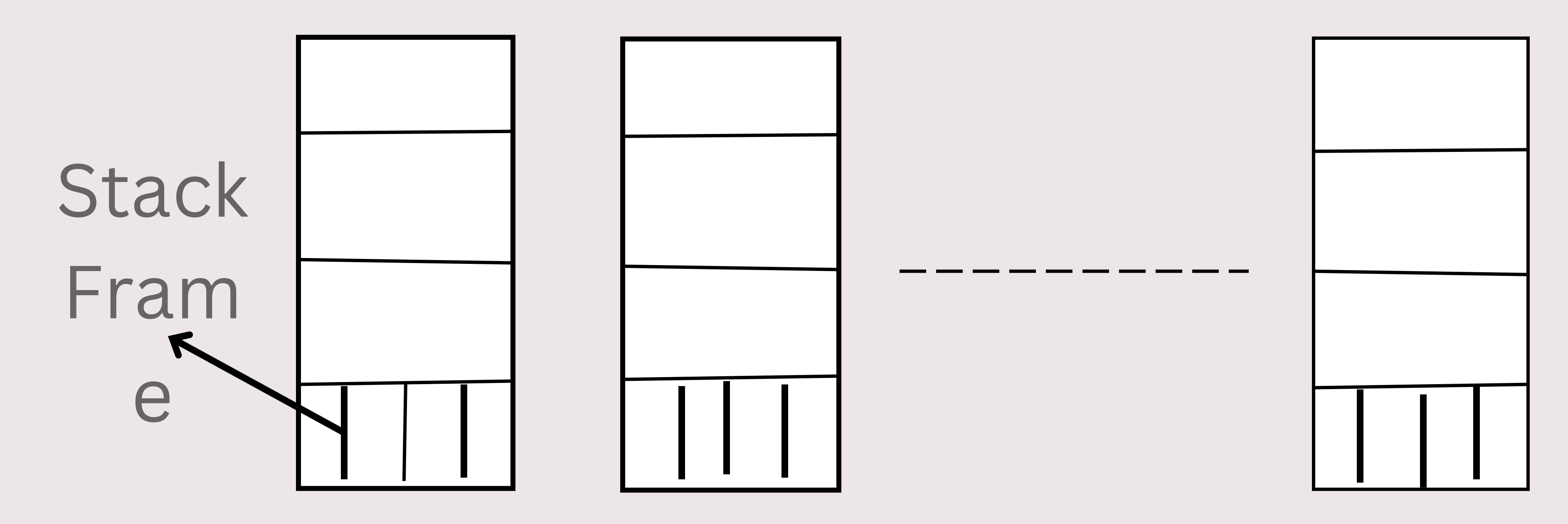
- It stores **objects, arrays, instance variable.**
- The Heap area is shared among all the threads running in the JVM. (Not thread safe.)
- Only one Heap area per JVM.

JVM-Class Loader

Stack Area



- It stores local variables, current running methods.
- For every thread, JVM creates one runtime stack.



- Each block of stack is called activation record/stack frame.
- Each frame contains: local variable, frame data and operand stack.

JVM-Class Loader

PC Register

- Method Area
- Heap Area
- Stack Area
- PC Register
- Native Method Stack

- It stores the current execution instruction. Once it completes, it automatically updates the next PC Register.
- Each thread has separate PC Registers.

JVM-Class Loader

Native Method Stack

Method Area

Heap Area

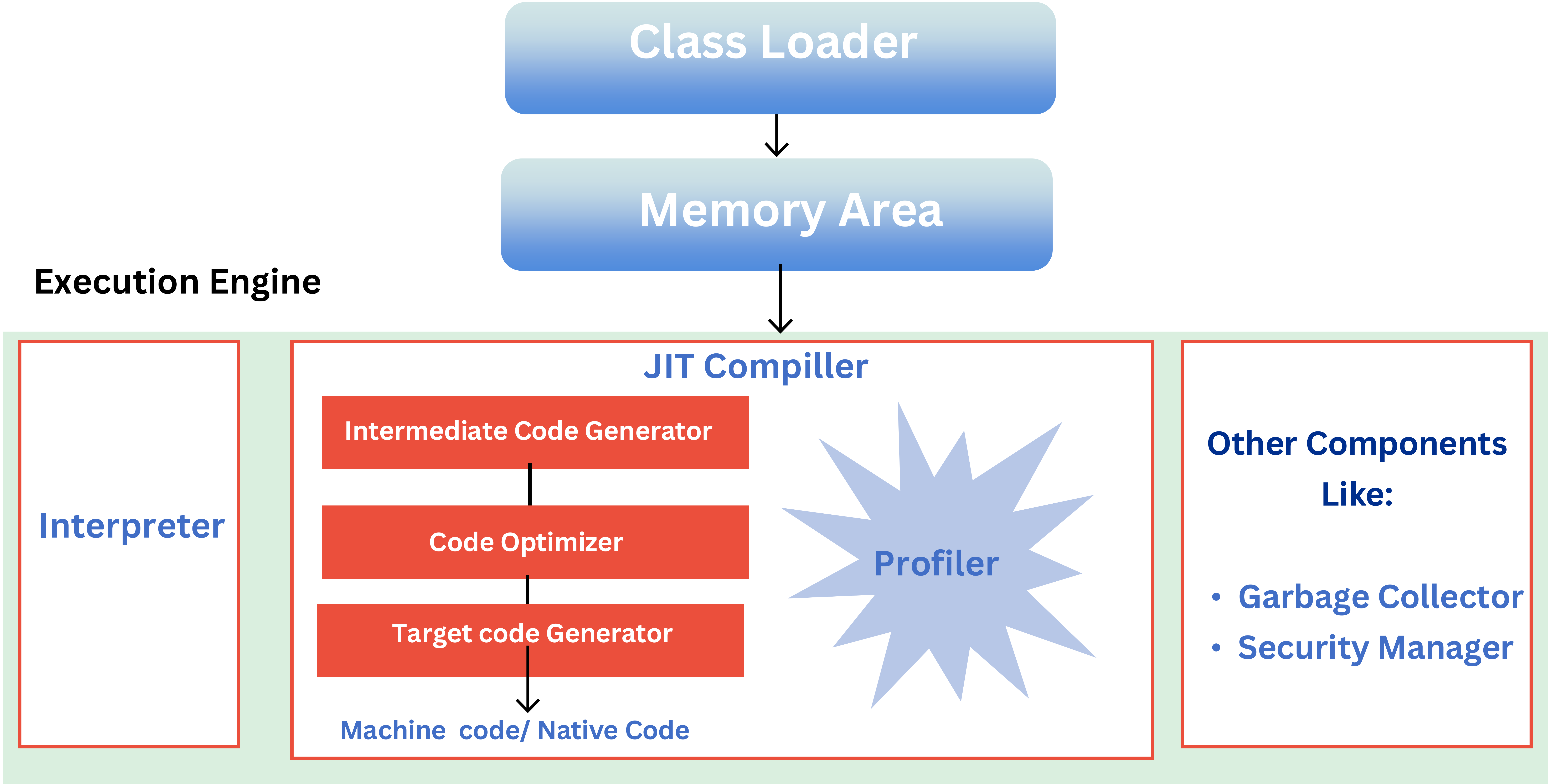
Stack Area

PC Register

Native Method Stack

- Memory used for native method execution.
- It is separate from the Java stack to handle native (non-Java) code.
- For every thread, a separate native stack is created.

JVM-Execution Engine



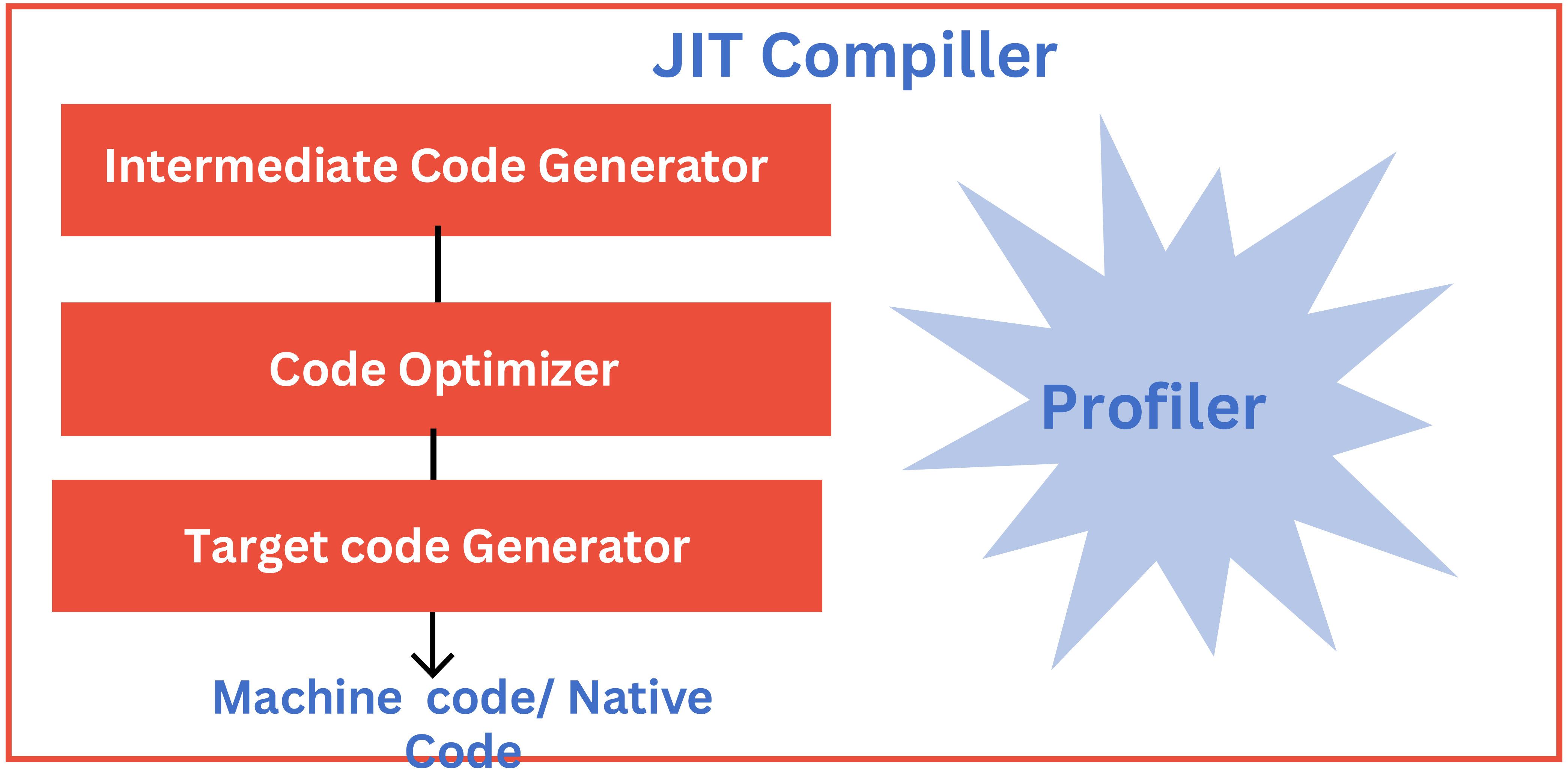
JVM-Class Loader

Interpreter

- It is responsible to **read byte code** and interpret it into machine code line by line.
- The problem with the interpreter is that it interprets every time, even for repeated method calls, **which affects performance.**
- To overcome this problem, the **JIT compiler** was introduced in version 1.1.

JVM-Execution Engine

JIT (Just In Time) Compiler



- The main purpose of the JIT compiler is to improve performance.
- It compiles the entire byte code and converts it into machine code.
- Whenever the interpreter sees repeated method calls, the JIT Compiler starts working on them.

Profiler :

It is responsible for identifying repeated method calls (Hotspot).

JVM-Execution Engine

Other Components :

There are several other components like the **Garbage Collector, Security Manager, etc.**

JVM-Execution Engine

Java Native Interface (JNI)

- JNI interacts with the Native Method Library and provides the native method library to the Execution Engine.
- In other words, JNI is responsible for providing native information to the JVM.

JVM- Architecture