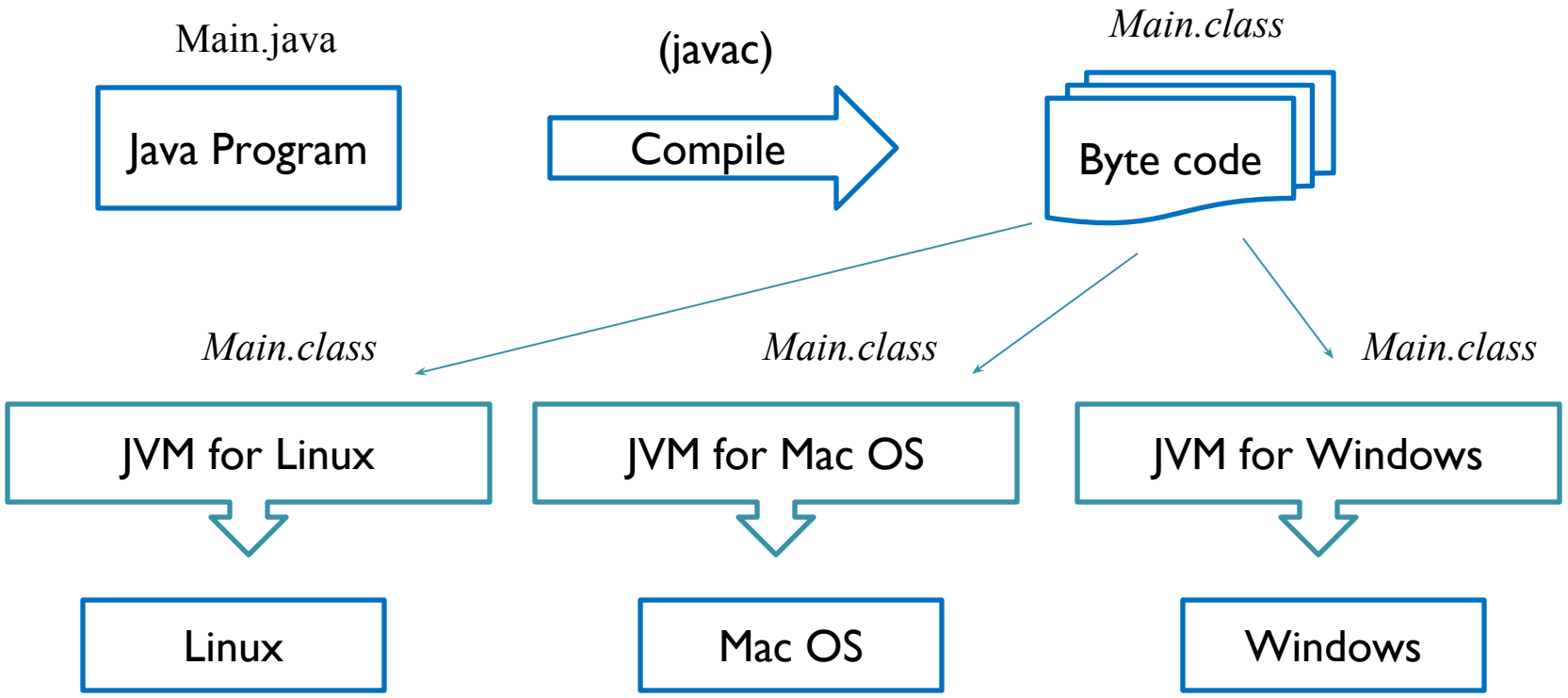
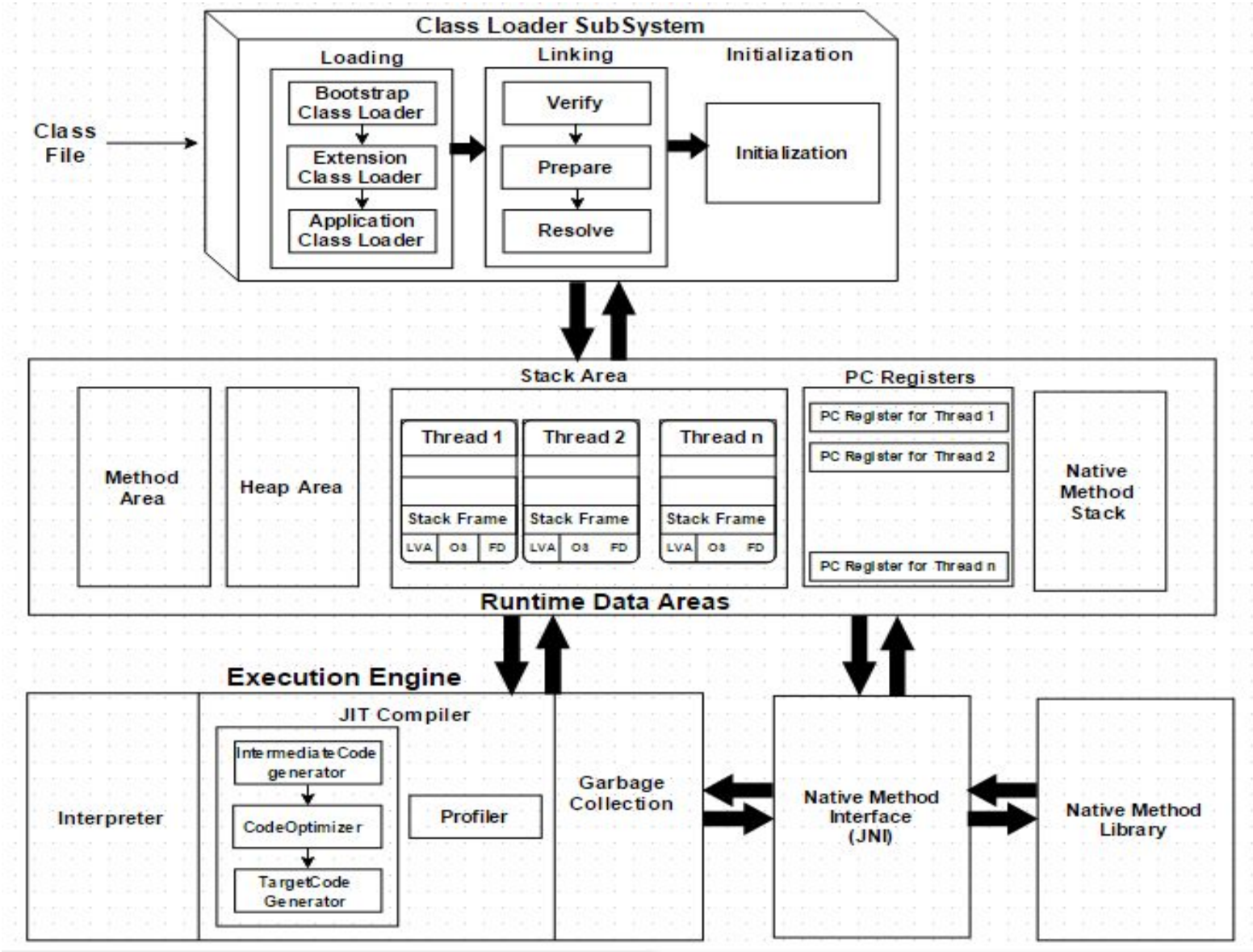
JVM (Java Virtual Machine)

The JVM is an interpretive computing engine that is responsible for running the byte codes in a compiled Java program.



JVM Architecture Diagram



Class Loader Sub System

Different part of Class Loader Sub System.

- Loading
  - **01. Bootstrap Class Loader**
  - **02. Extension Class Loader**
  - **03. Application Class Loader**
- Linking
  - **01. Verification**
  - **02. Preparation**
  - **03. Resolution**
- Initialization

Loading

- **Loading :** The Class loader reads the *.class* file, generate the corresponding binary data and save it in method area. For each *.class* file, JVM stores following information in method area.

- Fully qualified name of the loaded class and its immediate parent class.
- Whether *.class* file is related to Class or Interface or Enum
- Modifier, Variables and Method information etc.

After loading *.class* file, JVM creates an object of type Class to represent this file in the heap memory. This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc.

Note : JVM follow Delegation-Hierarchy principle to load classes.

Different Class Loaders

- **Bootstrap Class Loading:** The "*Bootstrap Class Loader*" loads the *rt.jar* file from '*jre\lib\rt.jar* ' which contains all class files of Java Standard Edition like java.lang package classes, java.util package classes etc.

- **Extension Class Loading:** The "*Extension Class Loader*" is responsible for loading all JAR files located at the '*jre\lib\ext* ' path. When a developer adds JARs manually, all those classes are loaded during this phase.

- **Application Class Loading**: The "*Application Class Loader*" loads all classes located in the application class path.

Linking

- **Verify** : Bytecode verifier will verify whether the generated bytecode is proper or not , else we will get Verification Error(java.lang.VerifyError)

- **Prepare** : For all static variables in bytecode , memory will be allocated and default values will be loaded.

- **Resolve** : Symbolic References of class will replaced with original References from method area.

Initialization

- In this phase, all static variables are assigned with their values defined in the code and static blocks are executed from top to bottom in a class and from parent to child in class hierarchy.

Runtime Data Area : (JVM Memory)

- **Method area :** In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

- **Heap area :** All the objects & corresponding instance , variable , array will be stored. There is also one Heap Area per JVM. It is also a shared resource.

- Where Heap & Method Area are not thread safe it shared resource for JVM

Runtime Data Area : (JVM Memory)

- **Stack Area** : For every thread new stack at runtime will be created. for every method call , one entry will be added in stack called **stack frame**.

- Local Variable will be stored in stack memory
- Thread safe
- Stack Frame is divide into three sub entities:
  - Local Variable Array
  - Operand Stack
  - Frame Data

Runtime Data Area : (JVM Memory)

- **PC Registers:** Each Thread will have register which stores the current executing operation.(once each instruction updating , PC register also will update next instruction)

- **Native Method Stack**: Holds native method information for every thread.

Execution Engine

- Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-

- ***Interpreter* :** It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.

- ***Just-In-Time Compiler(JIT)* :** It is used to increase efficiency of interpreter. It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls, JIT provide direct native code for that part so re-interpretation is not required, thus efficiency is improved.

- ***Garbage Collector* :** It destroy un-referenced objects.

Java Native Interface (JNI)

- It is an interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

Native Method Libraries

- It is a collection of the Native Libraries (C, C++) which are required by the Execution Engine.