SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution
Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

FUNDAMENTALS OF JAVA

III YEAR - IVISEM

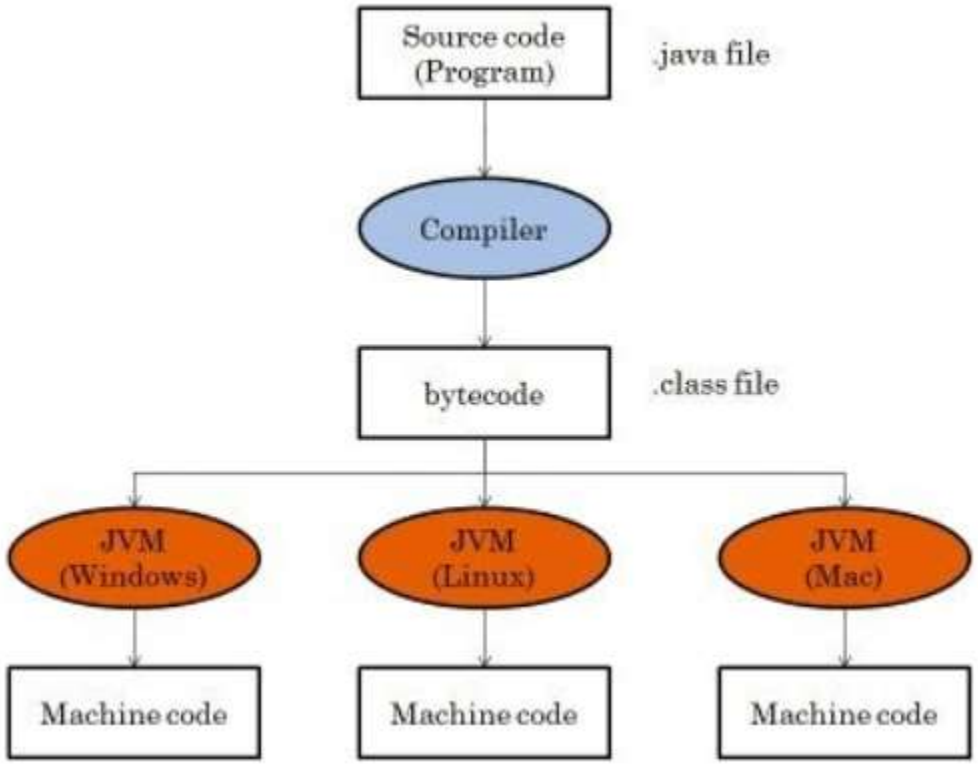
Topic - Byte Code and JVM

Java Bytecode

What is Java Bytecode?

Java bytecode is the instruction set for the Java Virtual Machine. It acts similar to an assembler which is an alias representation of a C++ code. As soon as a java program is compiled, java bytecode is generated. In more apt terms, java bytecode is the machine code in the form of a .class file. With the help of java bytecode we achieve platform independence in java.

How does it works?



Java Bytecode

Advantage of Java Bytecode

Platform independence is one of the soul reasons for which James Gosling started the formation of java and it is this implementation of bytecode which helps us to achieve this. Hence bytecode is a very important component of any java program.The set of instructions for the JVM may differ from system to system but all can interpret the bytecode. A point to keep in mind is that bytecodes are non-runnable codes and rely on the availability of an interpreter to execute and thus the JVM comes into play.

Bytecode is essentially the machine level language which runs on the Java Virtual Machine. Whenever a class is loaded, it gets a stream of bytecode per method of the class. Whenever that method is called during the execution of a program, the bytecode for that method gets invoked.Javac not only compiles the program but also generates the bytecode for the program. Thus, we have realized that the bytecode implementation makes Java a **platform-independent** language. This helps to add portability to Java which is lacking in languages like C or C++. Portability ensures that Java can be implemented on a wide array of platforms like desktops, mobile devices, severs and many more. Supporting this, Sun Microsystems captioned JAVA as *"write once, read anywhere" or "WORA"* in resonance to the bytecode interpretation.

Java Virtual Machine

What is JVM?

JVM (Java Virtual Machine) is an abstract machine that enables your computer to run a Java program.

When you run the Java program, Java compiler first compiles your Java code to bytecode. Then, the JVM translates bytecode into native machine code (set of instructions that a computer's CPU executes directly).

Java is a platform-independent language. It's because when you write Java code, it's ultimately written for JVM but not your physical machine (computer). Since JVM executes the Java bytecode which is platform-independent, Java is platform-independent.

Java Virtual Machine

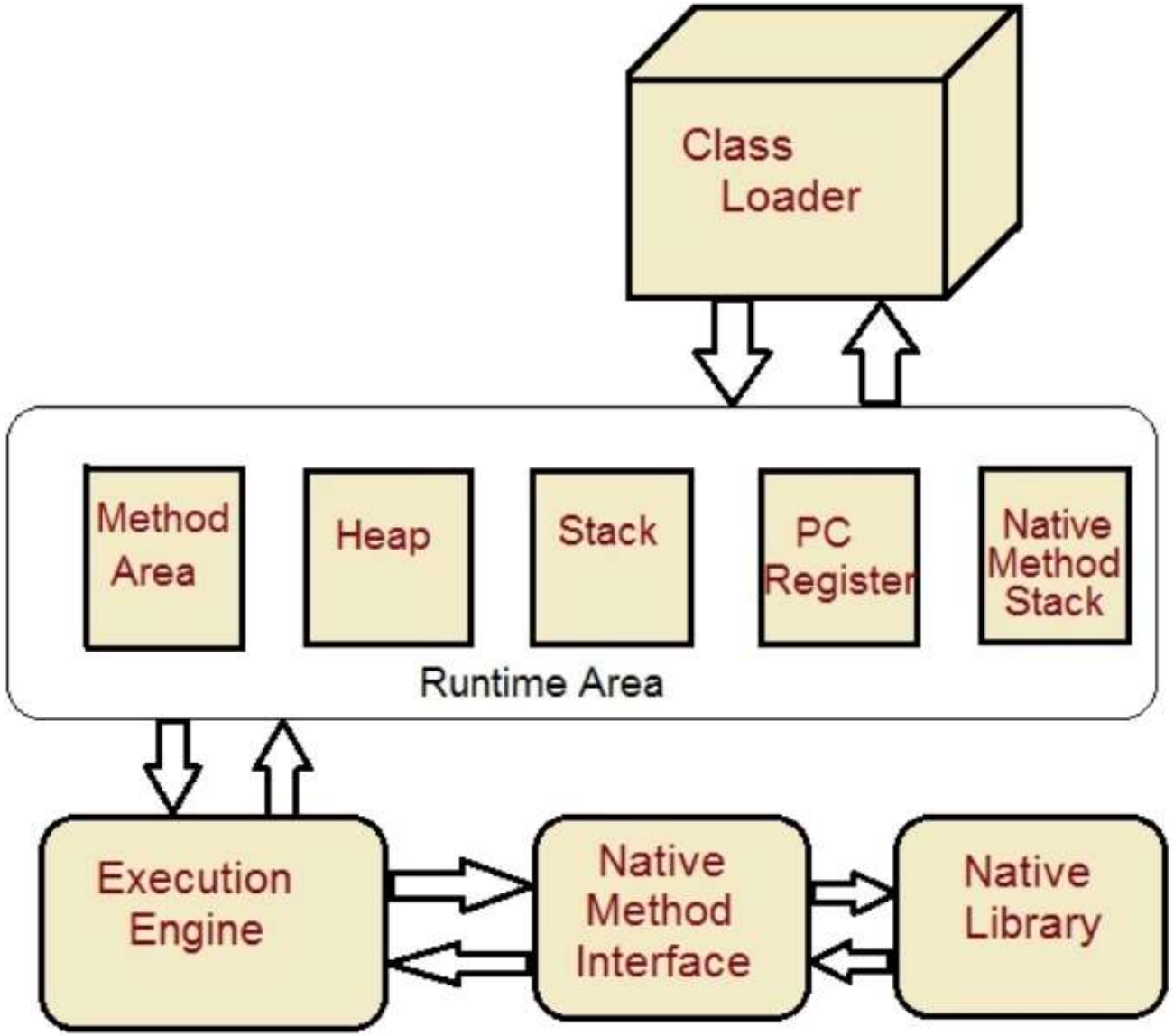
What is JVM?

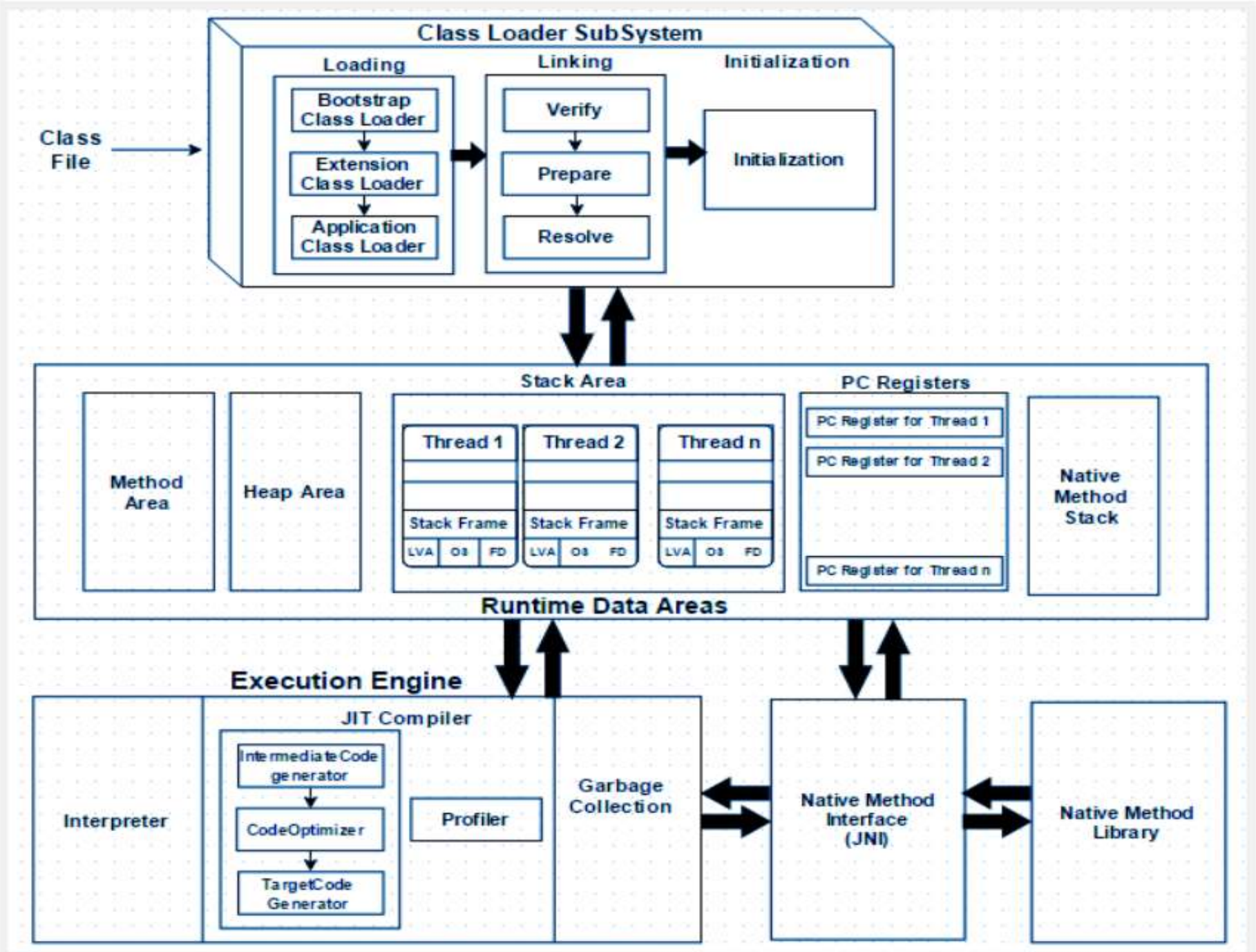
- JVM, i.e., Java Virtual Machine.
- JVM is the engine that drives the Java code.
- Mostly in other Programming Languages, compiler produce code for a particular system but Java compiler produce Bytecode for a Java Virtual Machine.
- When we compile a Java program, then bytecode is generated. Bytecode is the source code that can be used to run on any platform.
- Bytecode is an intermediary language between Java source and the host system.
- It is the medium which compiles Java code to bytecode which gets interpreted on a different machine and hence it makes it Platform/Operating system independent.

JVM's work can be explained in the following manner

- Reading Bytecode.
- Verifying bytecode.
- Linking the code with the library.

JVM Architecture




Class Loader SubSystem
Loading
Linking
Initialization
Bootstrap Class Loader
Extension Class Loader
Application Class Loader
Verify
Prepare
Resolve
Initialization
Class File
Stack Area
PC Registers
Method Area
Heap Area
Thread 1
Thread 2
Thread n
Stack Frame
LVA
OS
FD
PC Register for Thread 1
PC Register for Thread 2
PC Register for Thread n
Native Method Stack
Runtime Data Areas
Execution Engine
JIT Compiler
Intermediate Code generator
CodeOptimizer
Target Code Generator
Profiler
Interpreter
Garbage Collection
Native Method Interface (JNI)
Native Method Library

Java Virtual Machine

1. ClassLoader Subsystem

Java's dynamic class loading functionality is handled by the ClassLoader subsystem. It loads, links. and initializes the class file when it refers to a class for the first time at runtime, not compile time.

1.1 Loading

Classes will be loaded by this component. BootStrap ClassLoader, Extension ClassLoader, and Application ClassLoader are the three ClassLoaders that will help in achieving it.

1. **BootStrap ClassLoader** – Responsible for loading classes from the bootstrap classpath, nothing but **rt.jar.** Highest priority will be given to this loader.
2. **Extension ClassLoader** – Responsible for loading classes which are inside the ext folder **(jre\lib).**
3. **Application ClassLoader** –Responsible for loading Application Level Classpath, path mentioned Environment Variable, etc.

The above ClassLoaders will follow Delegation Hierarchy Algorithm while loading the class files.

Java Virtual Machine

1.2 Linking

1. **Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.
2. **Prepare** – For all static variables memory will be allocated and assigned with default values.
3. **Resolve** – All symbolic memory references are replaced with the original references from Method Area.

1.3 Initialization

This is the final phase of ClassLoading; here, all static variables will be assigned with the original values, and the static block will be executed.

Java Virtual Machine

2. Runtime Data Area

The Runtime Data Area is divided into five major components:

1. **Method Area** – All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
2. **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.
3. **Stack Area** – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three subentities:
    1. **Local Variable Array** – Related to the method how many local variables are involved and the corresponding values will be stored here.
    2. **Operand stack** – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
    3. **Frame data** – All symbols corresponding to the method is stored here. In the case of any **exception**, the catch block information will be maintained in the frame data.
4. **PC Registers** – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
5. **Native Method stacks** – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

Java Virtual Machine

3. Execution Engine

The bytecode, which is assigned to the **Runtime Data Area,** will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

1. **Interpreter** – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
2. **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
    1. **Intermediate Code Generator** – Produces intermediate code
    2. **Code Optimizer** – Responsible for optimizing the intermediate code generated above
    3. **Target Code Generator** – Responsible for Generating Machine Code or Native Code
    4. **Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.

Java Virtual Machine

3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling `System.gc()`, but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

Java Native Interface (JNI): JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

Native Method Libraries: This is a collection of the Native Libraries, which is required for the Execution Engine.

Java Runtime Environment

What is JRE?

JRE (Java Runtime Environment) is a software package that provides Java class libraries, Java Virtual Machine (JVM), and other components that are required to run Java applications.

JRE is the superset of JVM.

If you need to run Java programs, but not develop them, JRE is what you need. You can download JRE from Java SE Runtime Environment 8 Downloads page.

Java Development Kit

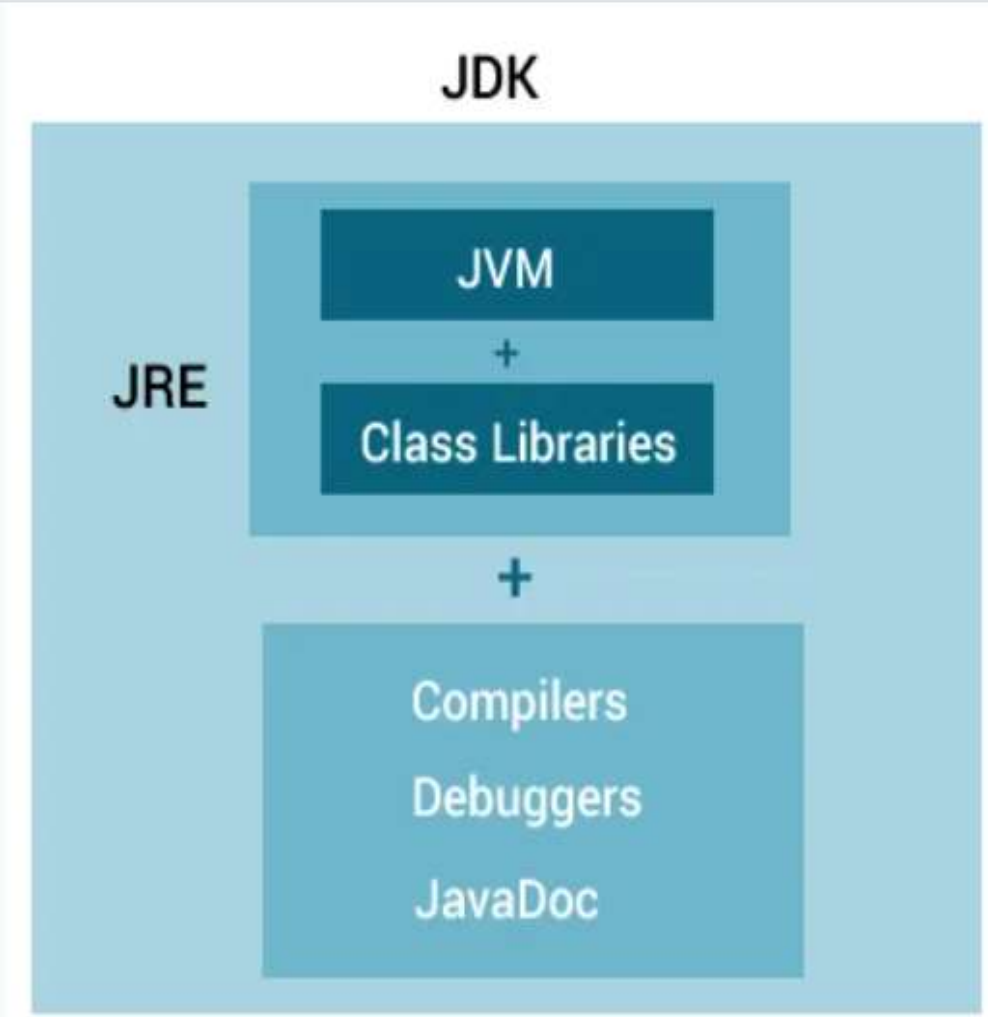
What is JDK?

JDK (Java Development Kit) is a software development kit required to develop applications in Java. When you download JDK, JRE is also downloaded with it.

In addition to JRE, JDK also contains a number of development tools (compilers, JavaDoc, Java Debugger, etc).

If you want to develop Java applications, download JDK.

Relationship between JVM, JRE, and JDK.



Relationship between JVM, JRE, and JDK