Contents

[**Javac(compiler), JVM, Java(interpreter) and JIT(compiler)** 1](#_Toc509945583)

|  |
| --- |
| **Javac(compiler)****--------------------------****JVM:** **Java (interpreter)** **JIT (compiler)** |
|  |
|  |
| How Java is platform independent  What gives Java its 'write once and run anywhere' nature? |
| * Platform Independence is also called build/write once, run anywhere. * When Java is compiled, it is not compiled into platform specific machine. The output is a class file, which contains an internal java representation called bytecode. * JVM converts bytecode to executable instructions (different in different operating systems). * There are different JVM's for different operating systems.   Example:  A JVM for windows is different from a JVM for mac. However, both the JVM's understand the bytecode and convert it to the executable code for the respective operating system. |
| What is ByteCode? |
| * Java bytecode is the instruction set of the Java virtual machine. * Each bytecode is composed of one, or in some cases two bytes that represent the instruction (opcode), along with zero or more bytes for passing parameters.   This helps minimize the size of class files that may be traveling across networks before being loaded by a JVM. |
| How Java compile the code? |
| * Suppose we write a java program and save it as “hello.java” in a folder somewhere on a hard disk. * The name of the java compiler is “javac”. For its compilation we have to write “javac hello.java” on the command line. * Java converts .java file into .class file as “hello.class” . This code is also known as a byte code because every instruction in this file is of 1 Byte. |
| Who execute the .class file?  What is JVM ? |
| Step1 : Before compile A.java:    Step2 : Compiling A.java file:    Step3 : After compile A.java:    Step4 : Now we have .class files , one for each class:   * Output of javac compiler (.class file) works as input into the Java Virtual Machine (JVM). * JVM is a virtual/abstract machine which provides the runtime environment in which java bytecode can be executed. * JVM is a specification. * JVMs are available for many hardware and software platforms (so JVM is platform dependent). |
| What is JRE? |
| * JRE can be considered as implementation of JVM. * The Java Runtime Environment (JRE) provides below items to run Java applets and applications:  1. libraries 2. Java Virtual Machine 3. other components  * In addition, two key deployment technologies are also part of the JRE:  1. Java Plug-in, which enables applets to run in popular browsers. 2. Java Web Start, which deploys standalone applications over a network.  * The JRE does not contain tools and utilities such as compilers or debuggers for developing applets and applications. |
| What is JDK? |
| * The Java Development Kit (JDK) is a superset of the JRE , * It contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications. |
| List out main components of JVM? |
| JVM is divided into three main subsystems:  1. Class Loader Subsystem  2. Runtime Data Area  3. Execution Engine |
| What is Classloader?  What is the responsibility of Class Loader Subsystem in JVM? |
| * Java's dynamic class loading functionality is handled by the class loader subsystem. * It load, links and initializes the class file when it refers to a class for the first time at runtime (not compile time).   [JVM Architecture Diagram](http://www.javainterviewpoint.com/wp-content/uploads/2016/01/JVM-Architecture.png) |
| How JVM load the class file using Class Loader Subsystem (in JVM)? |
| There are three class loaders which 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.      1. Extension ClassLoader – Responsible for loading classes which are inside ext folder (jre\lib)      1. Application ClassLoader – Responsible for loading Application Level Classpath , path mentioned Environment Variable etc. |
| How JVM link the class file using Class Loader Subsystem (in JVM)? |
| Class linking happens by completing below tasks:   1. Verify – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get 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. |
| What happens in initialization phase of the Class Loader Subsystem (in JVM)? |
| * This is the final phase of Class Loading, * Here all static variables will be assigned with the original values and static block will be executed. |
| What is the use of Runtime Data Area subsystem? |
| * To store Runtime Data. * Runtime Data Area is divided into 5 major components  1. Method Area 2. Heap Area 3. Stack Area : Stack Frame is divided into three sub-entities such as  * Local Variable Array * Operand stack * Frame data  1. PC Registers 2. Native Method stacks [JVM Architecture Diagram](http://www.javainterviewpoint.com/wp-content/uploads/2016/01/JVM-Architecture.png) |
| What “Method (Class) Area” stores in Runtime Data Area subsystem of JVM? |
| * All the Class level data will be stored here including static variables. * Method Area is one per JVM and it is a shared resource.   Since Method area is shares memory for multiple threads the data stored is not thread safe. |
| What “Heap Area” stores in Runtime Data Area subsystem of JVM? |
| * All the Objects and its corresponding instance variables and arrays will be stored here. * Heap Area is also one per JVM   Since Heap area shares memory for multiple threads the data stored is not thread safe. |
| What “Stack Area” stores in Runtime Data Area subsystem of JVM? |
| * In 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 as Stack Frame. * Stack Frame is divided into three sub-entities :  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 act 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 |
| What is the use of Execution Engine subsystem in JVM? |
| * 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. * Main component related to Execution Engine are as below:  1. Interpreter 2. JIT Compiler 3. Intermediate Code generator 4. Code Optimizer 5. Target Code Generator 6. Profiler 7. Garbage Collector 8. Java Native Interface (JNI)   [JVM Architecture Diagram](http://www.javainterviewpoint.com/wp-content/uploads/2016/01/JVM-Architecture.png) |
| What is the responsibility of Interpreter in Execution Engine subsystem of JVM? |
| * Interpreter – Reads the bytecode, interprets it and executes it one by one. * The interpreter interprets the bytecode faster but executes slowly. * The disadvantage of the interpreter is that when one method called multiple times, every time interpretation is required. |
| What is the responsibility of JIT Compiler in Execution Engine subsystem of JVM? |
| * JIT Compiler neutralizes the disadvantage of the Interpreter ( a single method called multiple times, each time interpretation is required ) * The Execution Engine will be using the help of Interpreter in converting but when it found repeated code it uses 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 – Code Optimizer is responsible for optimizing the intermediate code generated above 3. Target Code Generator – Target Code Generator is responsible for Generating Machine Code/ Native Code 4. Profiler – Profiler is a special component, it is responsible for finding the hotspots (i.e) Used to identify whether the method is called multiple time or not. |
| What is the responsibility of Garbage Collector in Execution Engine subsystem of JVM? |
| * Garbage Collector is a part of Execution Engine, it collects/removes the unreferenced objects. * Garbage Collection can be triggered by calling “System.gc()”, but the execution is not guaranteed. * Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform clean-up. |
| What is the responsibility of Java Native Interface (JNI) in Execution Engine subsystem of JVM? |
| * 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: It is a Collection of the Native Libraries which is required for the Execution Engine. |