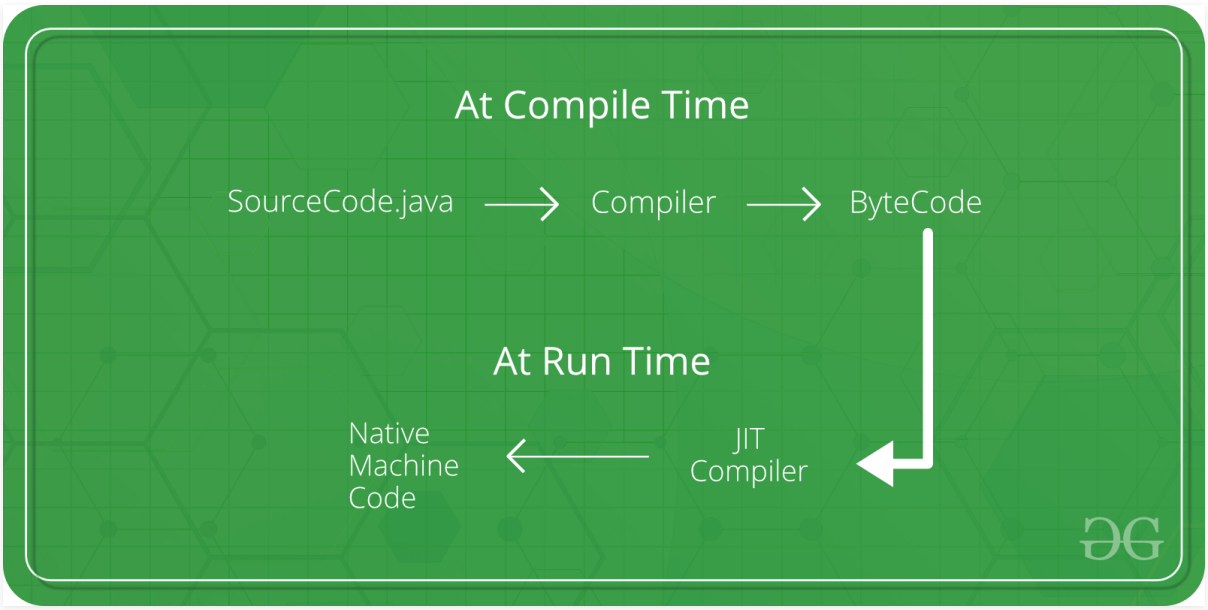
Difference between JIT and JVM in Java

Difficulty Level : Expert Last Updated : 09 Mar, 2021

Java Virtual Machine (JVM) is used in the java runtime environment(JRE). The original JVM was conceived as a bytecode interpreter. This may come as a bit of a surprise because of performance problems. Many modern languages are meant to be compiled into CPU-specific, executable code. The fact that the JVM executes a Java program, however, helps address the major issues associated with web-based applications.



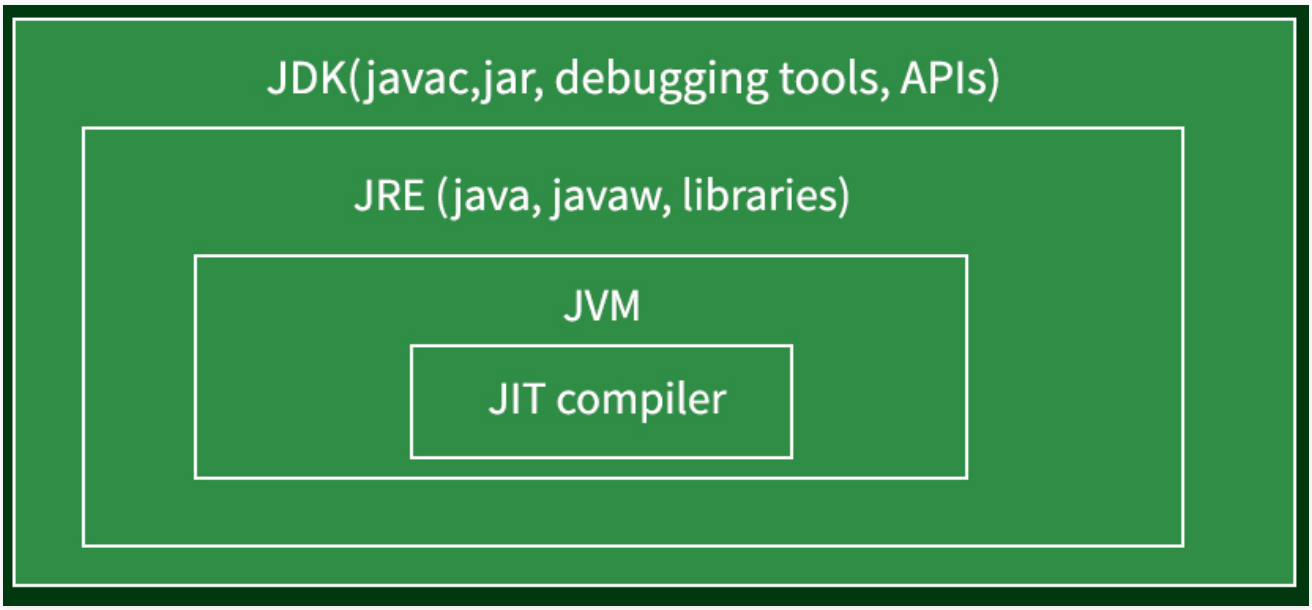
The fact that the JVM executes a Java program also helps to make it stable. Since the JVM is in charge, program execution is controlled by it. Therefore, it is possible for the JVM to build a limited execution area called a sandbox that contains the software, preventing the system from getting unlimited access. Protection is also improved by some limitations in the Java language that exists. Java's JVM architecture includes a class loader, execution engine, memory field, etc.

In order to understand differences, let's dig down to the components by illustrating the working of JVM alongside.

- **ClassLoader:** The class loader has the purpose of loading class files. It helps accomplish three main functions: Loading, Initialization, and Linking.
- **JVM language Stacks:** Java memory stores local variables, and partial results of a computation. Each thread has its own JVM stack, created as the thread is created. When the method is invoked, a new frame is created, and then removed.
- **Method Area:** JVM Method Area specializes in storing the metadata and code-behind files for Java applications.
- **PC Registers:** The Java Virtual Machine Instruction address currently being executed is saved

by PC registers. Each thread in Java has its own separate PC register.

- **Heap:** In a heap are saved all objects, arrays, and instance variables. This memory is shared between several threads.
- **Execution Engine:** It is a form of software used for the testing of software, hardware, or complete systems. The test execution engine never carries any information concerning the product being tested.
- **Native Method Libraries** which are the Executing Engine needs Native Libraries (C, C++) and the native method interface which is a framework for programming is the Native Method Interface. This enables the Java code that runs in a JVM to call libraries and native applications. Also, the native method stacks have a native code command depending on the native library. It assigns storage to native heaps or uses any stack type.



Just In Time(JIT) compiler

While Java was developed as an interpreted language, in order to improve performance, there is nothing about Java that prevents bytecode compilation into native code on the fly. For that reason, not long after Java's initial release, the HotSpot JVM was released. A just-in-time (JIT) bytecode compiler is included in HotSpot. A Just In Time(JIT) compiler is part of the JVM and on a piece-by-piece demand basis, selected portions of bytecode are compiled into executable code in real-time. That is, as is necessary during execution, a JIT compiler compiles code. In addition, not all bytecode sequences are compiled, only those that will benefit from the compilation. The just-in-time method, however, still yields a major boost in inefficiency. The portability and safety function still exists even though dynamic compilation is applied to bytecode since the JVM is still in control of the execution environment.

In order to understand differences, let's dig down to the components by illustrating the working of JIT alongside.

Interpreting the bytecode, the standard implementation of the JVM slows the execution of the programs. JIT compilers interact with JVM at runtime to improve performance and compile appropriate bytecode sequences into native machine code.

Hardware is interpreting the code instead of JVM (Java Virtual Machine). This can lead to performance gains in the speed of execution. This can be done per-file, per-function, or maybe on any arbitrary code fragment; the code is often compiled when it's close to being executed (hence the name "just-in-time"), and then cached and reused later without having to be recompiled. It performs many optimizations: data analysis, translation from stack operations to registry operations, reduction of memory access by registry allocation, elimination of common sub-expressions.

Hence, from the above knowledge, we landed on the conclusive differences between them as mentioned in the table below:

JVM	JIT
JVM stands for Java Virtual Machine.	JIT stands for Just-in-time compilation.
JVM was introduced for managing system memory and providing a transportable execution environment for Java-based applications	JIT was invented to improve the performance of JVM after many years of its initial release.
JVM consists of many other components like stack area, heap area, etc.	JIT is one of the components of JVM.
JVM compiles complete byte code to machine code.	JIT compiles only the reusable byte code to machine code.
JVM provides platform independence.	JIT improves the performance of JVM.