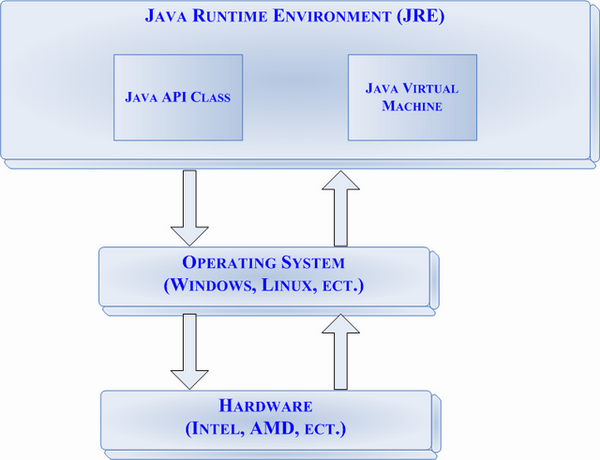
**Java Architecture**

In high-level programming languages such as C and C++, we write a program in a human-readable format, and a program called a compiler translates it to a binary format called executable code that the computer can understand and execute. The executable code depends upon the computer machine that we use to execute our program; it is machine dependent. In Java, this process of writing to executing a program is very similar, but with one important difference that allows us to write Java programs that are machine independent.

Using an interpreter, all Java programs are compiled to an intermediate level called byte code. We can run the compiled byte code on any computer with the Java runtime environment installed on it. The runtime environment consists of a virtual machine and its supporting code

**JVM is an EMULATION**

The difficult part of creating Java byte code is that the source code is compiled for a machine that does not exist. This machine is called the Java Virtual Machine, and it exists only in the memory of our computer. Fooling the Java compiler into creating byte code for a nonexistent machine is only one-half of the ingenious process that makes the Java architecture neutral. The Java interpreter must also make our computer and the byte code file believe they are running on a real machine. It does this by acting as the intermediary between the Virtual Machine and our real machine. (See figure below.)



**1. Compilation and interpretation in Java**

Java combines both the approaches of compilation and interpretation. First, java compiler compiles the source code into bytecode. At the run time, Java Virtual Machine (JVM) interprets this bytecode and generates machine code which will be directly executed by the machine in which java program runs. So java is both compiled and interpreted language.

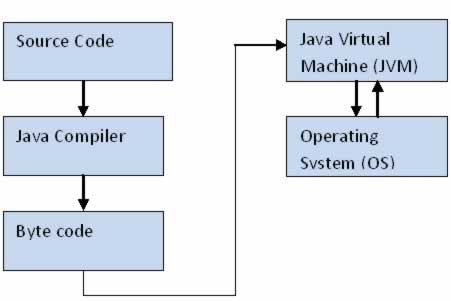
****

Figure 1: Java Architecture

## 2. Java Virtual Machine (JVM)

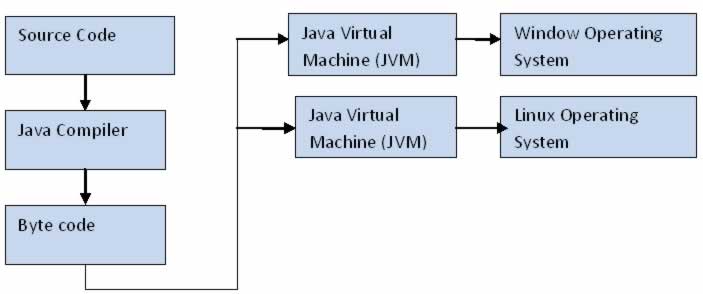
JVM is a component which provides an environment for running Java programs. JVM interprets the bytecode into machine code which will be executed in the machine in which Java program runs.

## 3. Why Java is Platform Independent?

Platform independence is one of the main advantages of Java. In another words, java is portable because the same java program can be executed in multiple platforms without making any changes in the source code. You just need to write the java code for one platform and the same program will run in any platforms. But how does Java make this possible?

As we discussed early, first the Java code is compiled by the Java compiler and generates the bytecode.  This bytecode will be stored in class files**.** Java Virtual Machine (JVM) is unique for each platform.   Though JVM is unique for each platform, all interpret the same bytecode and convert it into machine code required for its own platform and this machine code will be directly executed by the machine in which java program runs.  This makes Java platform independent and portable.

Let’s make it more clear with the help of the following diagram.  Here the same compiled Java bytecode is interpreted by two different JVMS to make it run in Windows and Linux platforms.



## 4. Java Runtime Environment (JRE) and Java Architecture in Detail

Java Runtime Environment contains JVM, class libraries and other supporting components.

As you know the Java source code is compiled into bytecode by Java compiler. This bytecode will be stored in class files. During runtime, this bytecode will be loaded, verified and   JVM interprets the bytecode into machine code which will be executed in the machine in which the Java program runs.

A Java Runtime Environment performs the following main tasks respectively.

1. Loads the class

This is done by the class loader

2. Verifies the bytecode

This is done by bytecode verifier.

3. Interprets the bytecode

This is done by the JVM

These tasks are described in detail in the subsequent sessions.A detailed Java architecture can be drawn as given below.

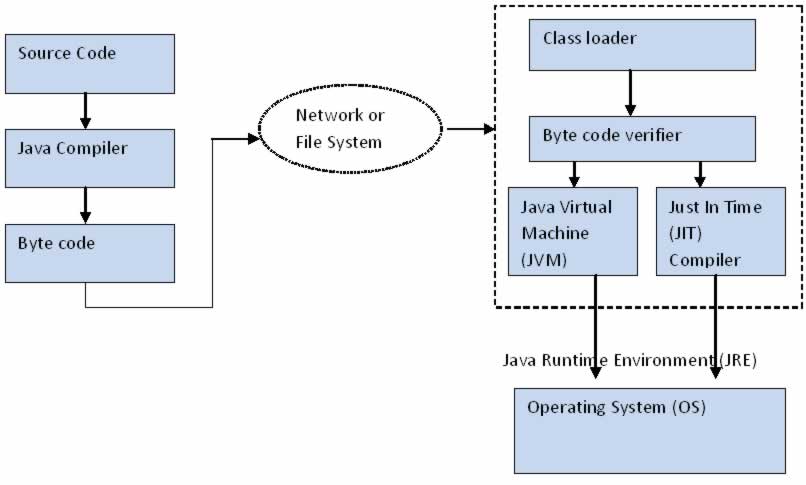


Figure 3: Java Architecture in Detail

## 4.1. Class loader

Class loader loads all the class files required to execute the program. Class loader makes the program secure by separating the namespace for the classes obtained through the network from the classes available locally. Once the bytecode is loaded successfully, then next step is bytecode verification by bytecode verifier.

## 4.2. Byte code verifier

The bytecode verifier verifies the byte code to see if any security problems are there in the code. It checks the byte code and ensures the followings.

1. The code follows JVM specifications.

2. There is no unauthorized access to memory.

3. The code does not cause any stack overflows.

4. There are no illegal data conversions in the code such as float to object references.

Once this code is verified and proven that there is no security issues with the code, JVM will convert the byte code into machine code which will be directly executed by the machine in which the Java program runs.

## 4.3. Just in Time Compiler

You might have noticed the component “Just in Time” (JIT) compiler in Figure 3. This is a component which helps the program execution to happen faster. How? Let’s see in detail.

As we discussed earlier when the Java program is executed, the byte code is interpreted by JVM.  But this interpretation is a slower process. To overcome this difficulty, JRE include the component JIT compiler. JIT makes the execution faster.

If the JIT Compiler library exists, when a particular bytecode is executed first time, JIT complier compiles it into native machine code which can be directly executed by the machine in which the Java program runs. Once the byte code is recompiled by JIT compiler, the execution time needed will be much lesser. This compilation happens when the byte code is about to be executed and hence the name “Just in Time”.

Once the byte code is compiled into that particular machine code, it is cached by the JIT compiler and will be reused for the future needs. Hence the main performance improvement by using JIT compiler can be seen when the same code is executed again and again because JIT make use of the machine code which is cached and stored.

## 5. W****hy Java is Secure?****

As you have noticed in the prior session “Java Runtime Environment (JRE) and Java Architecture in Detail”, the byte code is inspected carefully before execution by Java Runtime Environment (JRE).  This is mainly done by the “Class loader” and “Byte code verifier”. Hence a high level of security is achieved.

## 6. Garbage Collection

Garbage collection is a process by which Java achieves better memory management. As you know, in object oriented programming, objects communicate to each other by passing messages. (If you are not clear about the concepts of objects, please read the prior chapter before continuing in this session).

Whenever an object is created, there will be some memory allocated for this object. This memory will remain as allocated until there are some references to this object. When there is no reference to this object, Java will assume that this object is not used anymore.  When garbage collection process happens, these objects will be destroyed and memory will be reclaimed.

Garbage collection happens automatically. There is no way that you can force garbage collection to happen. There are two methods “System.gc()” and “Runtime.gc()” through which you can make request for garbage collation. But calling these methods also will not force garbage collection to happen and you cannot make sure when this garbage collection will happen.