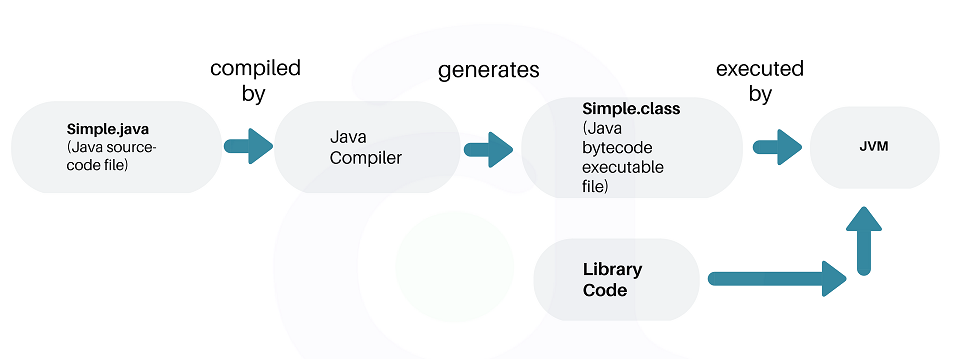
# 1. Execution of a Java Program

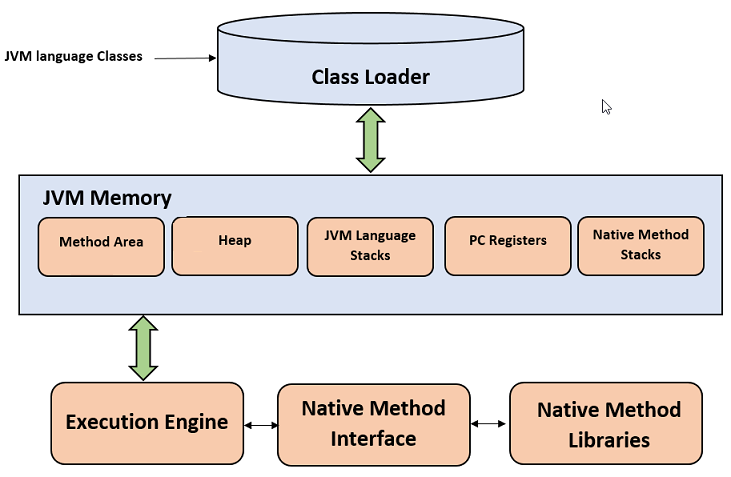


# 2. What is JVM?

**Java Virtual machine** (JVM) is the virtual machine that runs the Java bytecodes. You get this bytecode by compiling the .java files into .class files. .class files contain the bytecodes understood by the JVM.

Byte code is platform independent. JVM platform dependent.

#### 2.1. JVM Architecture



##### 2.1.1. Class Loader

The class loader is a subsystem used for loading class files. It performs three major functions i.e. class loading, linking, and initialization.

##### Loading

* + To load classes, JVM has 3 kind of class loaders. **Bootstrap**, **extension** and **application** class loader.
  + When loading a class file, JVM finds out a dependency for some arbitrary class XYZ.class.
  + First bootstrap class loader tries to find the class. It scans the rt.jar file in JRE lib folder.
  + If class is not found then extension class loader searches the class file in inside **jre\lib\ext** folder.
  + Again if class is not found then application classloader searches all the Jar files and classes in CLASSPATH environment variable of system.
  + If class is found by any loader then class is loaded by class loader; else ClassNotFoundException is thrown.

##### Linking

After class is loaded by the classloader, linking is performed. A **bytecode verifier** will verify whether the generated bytecode is proper or not if verification fails we will get a verification error. It also performs the memory allocation to static variables and methods found in the class.

##### Initialization

This is the final phase of class loading, here all static variable will be assigned with the original values and the static blocks will be executed.

##### 2.1.2. JVM Memory Areas

The memory area inside JVM is divided into multiple parts to store specific parts of application data.

* **Method Are**a stores class structures like metadata, the constant runtime pool, and the code for methods.
* **Heap** stores all objects that are created during application execution.
* **Stacks** store local variables, and intermediate results. All such variables are local to the thread by which they are created. Each thread has its own JVM stack, created simultaneously as the thread is created. So all such local variable are called **thread-local variables**.
* **PC register** store the physical memory address of the statements which is currently executing. In Java, each thread has its separate PC register.
* Java supports and uses **native code** as well. Many low level code is written in languages like C and C++. Native method stacks hold the instruction of native code.

Static variables:

First, static member variables are stored in the Permanent Generation area of heap.

When we create a static variable or method it is stored in the special area on heap: **PermGen(Permanent Generation)**, where it lays down with all the data applying to classes(non-instance data). Starting from Java 8 the PermGen became**- Metaspace**. The difference is that Metaspace is auto-growing space, while PermGen has a fixed Max size, and this space is shared among all of the instances. Plus the Metaspace is a part of a Native Memory and not JVM Memory.

Java Memory : <https://www.baeldung.com/java-stack-heap>

#### 2.2. JVM Execution Engine

All code assigned to JVM is executed by an **execution engine**. The execution engine reads the byte code and executes one by one. It uses two inbuilt *interpreter* and *JIT compiler* **to convert the bytecode to machine code and execute it**.

#### 2.2.1. Interpreter

A JVM interpreter pretty much converts each byte-code instruction to corresponding native instruction by looking up a predefined JVM-instruction to machine instruction mapping. It **directly** executes the bytecode and does not perform any optimization.

#### 2.2.2. JIT Compiler

**To improve performance**, JIT compilers interact with the JVM at runtime and compile appropriate bytecode sequences into native machine code. Typically, the JIT compiler takes a block of code (not one statement at a time as interpreter), optimize the code and then translates it to an optimized machine code.

The **JIT compiler is enabled by default**. You can disable the JIT compiler, in which case the entire Java program will be interpreted. Disabling the JIT compiler is not recommended except to diagnose or workaround JIT compilation problems.

# 3. What is JRE?

The Java Runtime Environment (JRE) is a software package which bundles the libraries (jars) and the Java Virtual Machine, and other components to run applications written in the Java. JVM is just a part of JRE distributions.

To execute any Java application, you need JRE installed in the machine. It’s minimum requirement to execute Java applications on any machine.

JRE bundles the following components –

1. DLL files used by the Java HotSpot Client Virtual Machine.
2. DLL files used by the Java HotSpot Server Virtual Machine.
3. Code libraries, property settings, and resource files used by the Java runtime environment. e.g. rt.jar and charsets.jar.
4. Java extension files such as localedata.jar.
5. Contains files used for security management. These include the security policy (java.policy) and security properties (java.security) files.
6. Jar files containing support classes for applets.
7. Contains TrueType font files for use by the platform.

# 4. What is JDK?

JDK is a superset of JRE. JDK contains everything that JRE has along with development tools for developing, debugging, and monitoring Java applications. You need JDK when you need to develop Java applications.

Few important components shipped with JDKs are as follows:

* appletviewer – this tool can be used to run and debug Java applets without a web browser
* apt – the annotation-processing tool
* extcheck – a utility that detects JAR file conflicts
* javadoc – the documentation generator, which automatically generates documentation from source code comments
* jar – the archiver, which packages related class libraries into a single JAR file. This tool also helps manage JAR files
* jarsigner – the jar signing and verification tool
* javap – the class file disassembler
* javaws – the Java Web Start launcher for JNLP applications
* JConsole – Java Monitoring and Management Console
* jhat – Java Heap Analysis Tool
* jrunscript – Java command-line script shell
* jstack – utility that prints Java stack traces of Java threads
* keytool – tool for manipulating the keystore
* policytool – the policy creation and management tool
* xjc – Part of the Java API for XML Binding (JAXB) API. It accepts an XML schema and generates Java classes

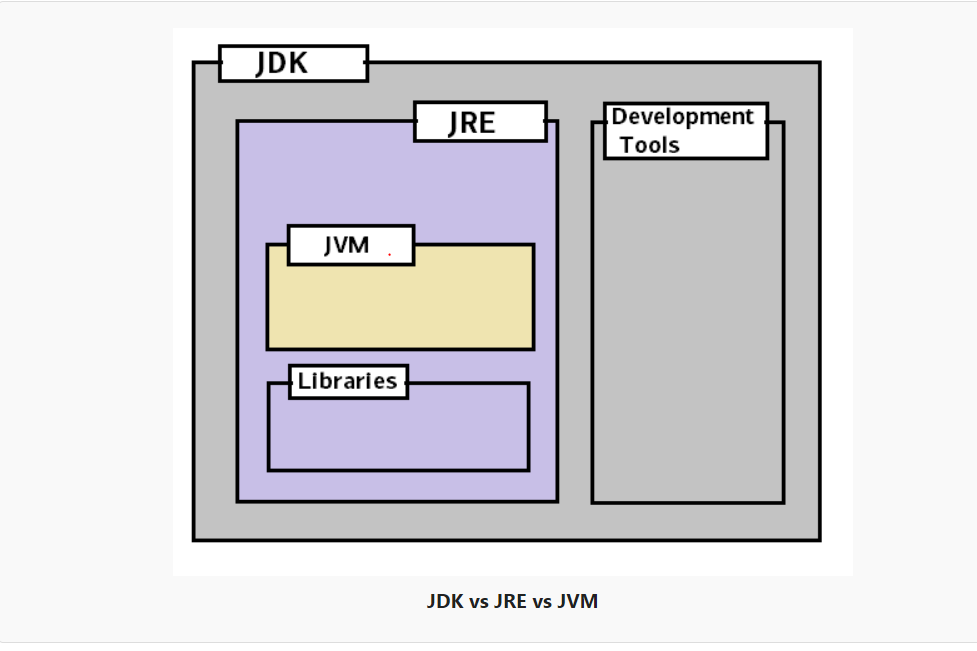
Same as JREs, JDKs are also platform dependent. So take care when you download the JDK package for your machine.

# 5. Difference between JDK, JRE and JVM

Based on the above discussions, we can draw a relationship between these three as below –

JRE = JVM + libraries to run Java application

JDK = JRE + tools to develop Java Application



In short, if you are a Java application developer who writes code, you will need JDK installed in your machine. But, if you only want to run applications built in Java, you only need JRE installed into your machine.



# Following are the notable features of Java:

### Object Oriented

In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

#### Platform Independent

Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform-independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.

### Simple

Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.

#### Secure

With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

### Architecture-neutral

Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.

### Portable

Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. The compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.

### Robust

Java makes an effort to eliminate error-prone situations by emphasizing mainly on compile time error checking and runtime checking.

### Multithreaded

With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.

### Interpreted

Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.

### High Performance

With the use of Just-In-Time compilers, Java enables high performance.

### Distributed

Java is designed for the distributed environment of the internet.

### Dynamic

Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry an extensive amount of run-time information that can be used to verify and resolve accesses to objects at run-time.

**Compiling and running a java program is very easy after JDK installation.**

Following are the steps −

* Open a command prompt window and go to the directory where you saved the java program (MyFirstJavaProgram.java). Assume it's C:\.
* Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption: The path variable is set).
* Now, type ' java MyFirstJavaProgram ' to run your program.
* You will be able to see the result printed on the window.