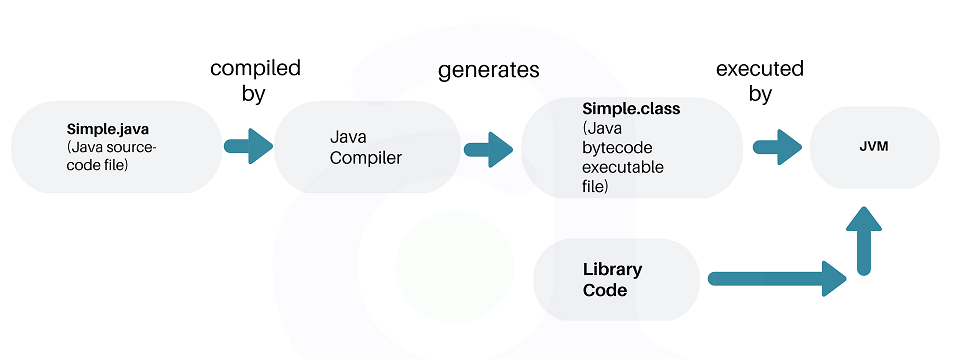
1. We write the Java source code in Simple.Java file using an editor or IDE (**integrated development environment**) e.g. *Eclipse* or *IntelliJ Idea*.
2. Program has to be compiled into bytecode. Java compiler (javac) compiles the sourcecode to Simple.class file.
3. This class file can be executed in any platform/OS by JVM (**Java virtual machine**).
4. JVM translates bytecode into native machine code which machines can execute.



**What is JVM?**

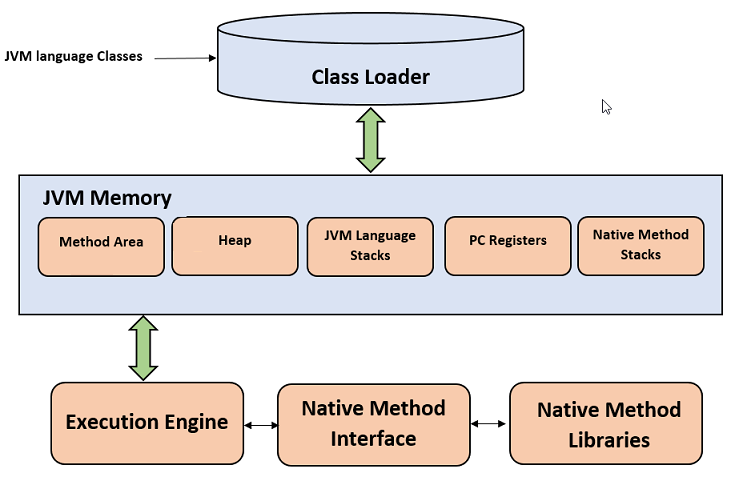
**Java Virtual machine** (JVM) is the virtual machine that runs the Java bytecodes. You get this bytecode by compiling the .java files into .class files. .class files contain the bytecodes understood by the JVM.

In the real world, JVM is a specification that provides a runtime environment in which Java bytecode can be executed.

JVM delivers the optimal performance for Java applications using many advanced techniques, incorporating a state-of-the-art memory model, **garbage collector**, and **adaptive optimizer**.

The JVM is called **virtual** because it provides a machine interface that does not depend on the underlying operating system and machine hardware architecture. This independence from hardware and the operating system is a cornerstone of the write-once-run-anywhere value of Java programs.

**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.

1. **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.
2. **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.

1. **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.

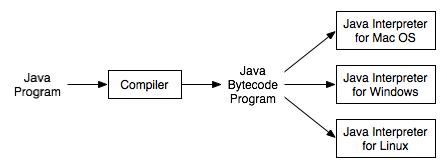
**2.1.2. JVM Memory Areas**

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. Each thread has its own JVM stack, created simultaneously as the thread is created. **PC register** store the physical memory address of the statements which is currently executing. In Java, each thread has its separate PC register.

**2.2. JVM Execution Engine**

All code is executed by an **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**.

Platform Specific Interpreters

With JVM, both interpreter and compiler produce native code. The difference is in how they generate the native code, how optimized it is as well how costly the optimization is.

**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, JIT compiler takes a block of code (not one statement at a time as interpreter), optimize the code and then translate it to 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 work around JIT compilation problems.

**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.

JREs can be downloaded as part of JDKs or you can download them separately. JREs are platform dependent. It means that based on the type of machine (OS and architecture), you will have to select the JRE bundle to import and install.

For example, you cannot install a 64-bit JRE distribution on 32-bit machine. Similarly, JRE distribution for *Windows* will not work in *Linux*; and vice-versa.

**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

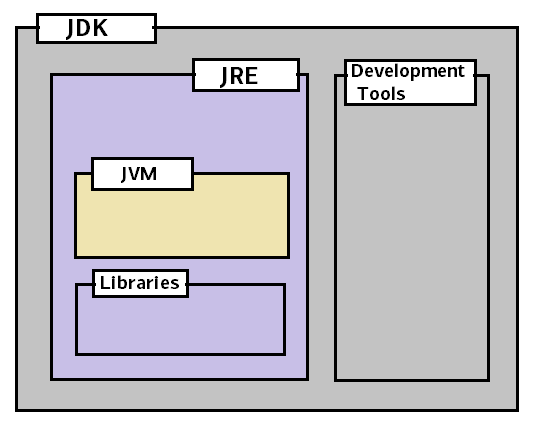
Same as JREs, JDKs are also platform dependent. So take care when you download the JDK package for your machine.

**5. Differences 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.

JDK vs JRE vs JVM

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.