Features of Java

The prime reason behind creation of Java was to bring portability and security feature into a computer language.

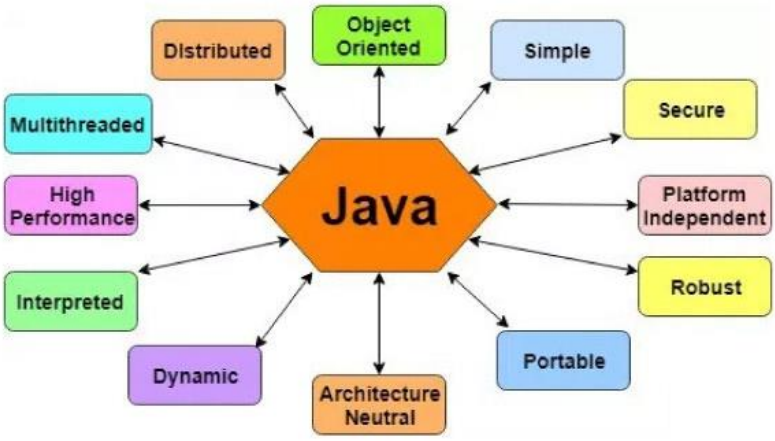
The primary objective of Java programming language creation was to make it portable, simple and secure programming language.

Apart from this, there are also some excellent features which play an important role in the popularity of this language.

The features of Java are also known as java buzzwords.

Those features are:

1. Simple
2. Object-Oriented
3. Platform independent
4. Portable
5. Secured
6. Robust
7. Architecture neutral
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed
12. Dynamic



1) Simple

- Java is easy to learn and its syntax is quite simple, clean and easy to understand. Java syntax is based on C++ (so easier for programmers to learn it after C++).

- The confusing and ambiguous concepts of C++ are either left out in Java or they have been re-implemented in a cleaner way.

- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Eg: Pointers and Operator Overloading are not there in java but were an important part of C++.

2) Object Oriented

Java is an object-oriented programming language. Everything in Java is an object which has some data and behaviour.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

3) Platform Independent

Java is platform independent because it is different from other languages like C, C++, etc.

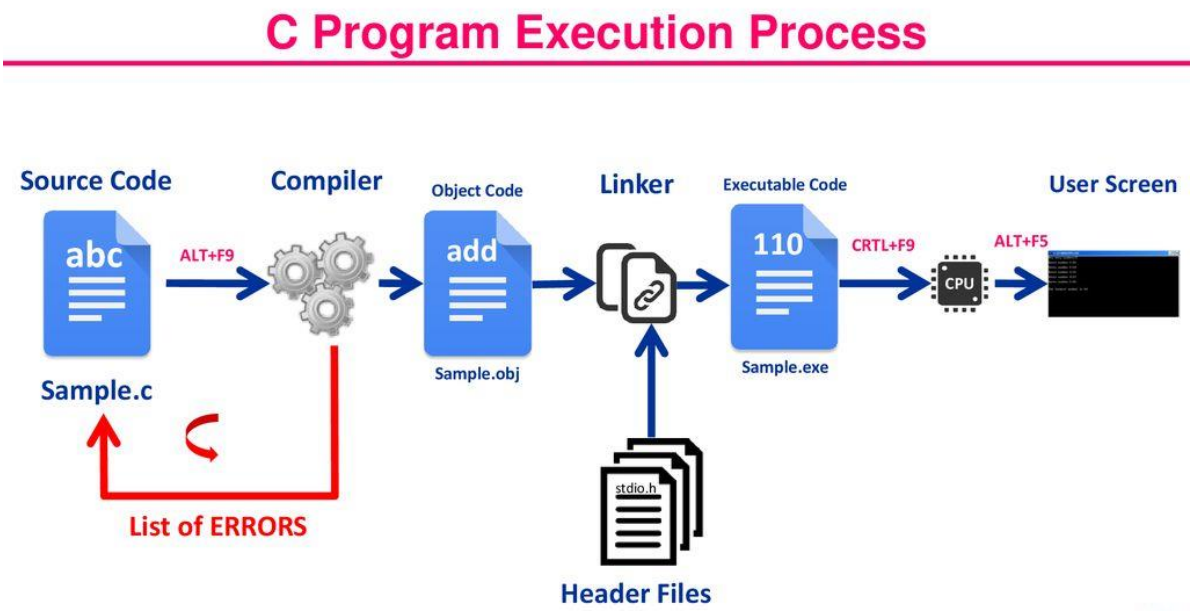
Unlike other programming languages such as C, C++ etc. which are compiled into platform specific machines. Java is guaranteed to be write-once, run-anywhere language.

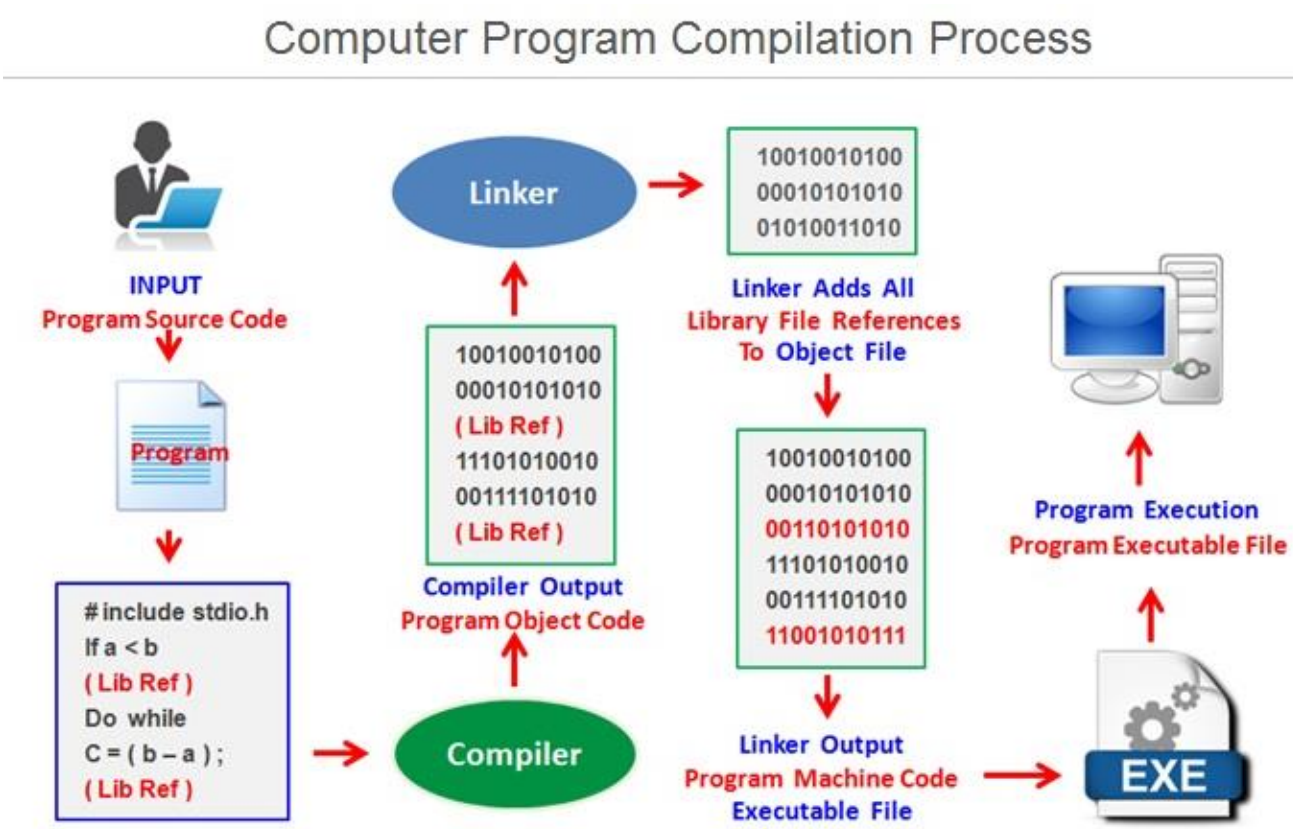
A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

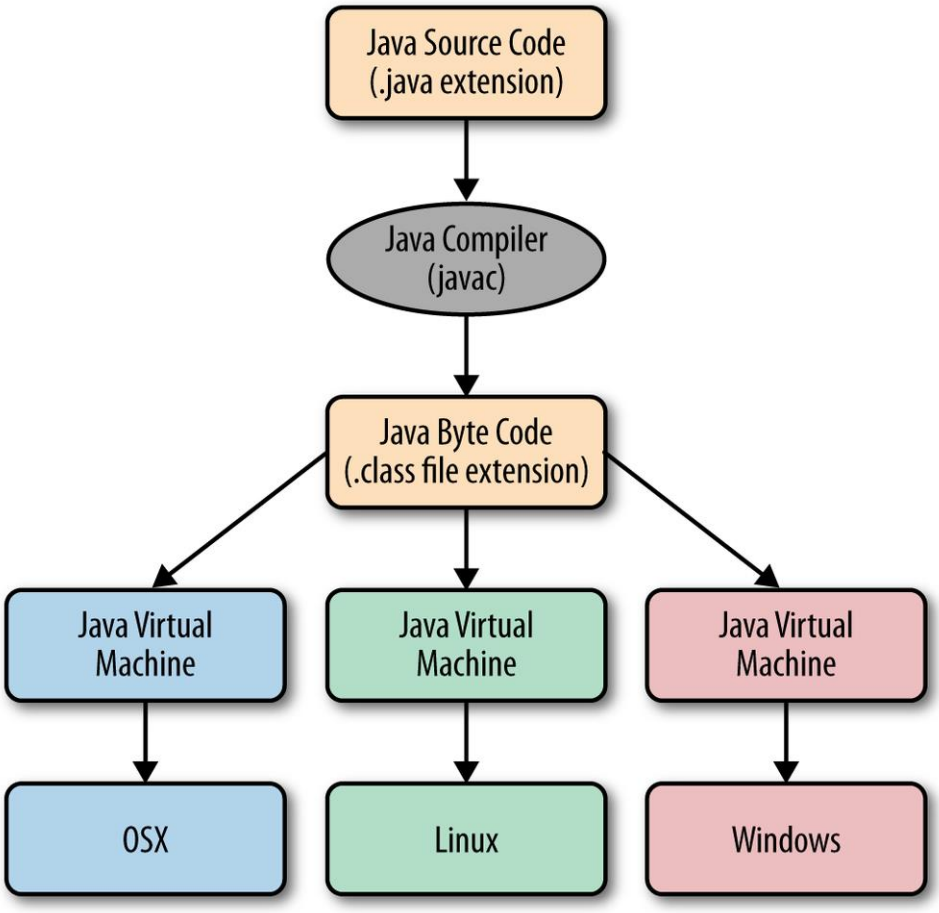
On compilation Java program is compiled into bytecode. This bytecode is platform independent and can be run on any machine, plus this bytecode format also provide security. Any machine with Java Runtime Environment can run Java Programs.

Example:





Java Program Compilation Process:



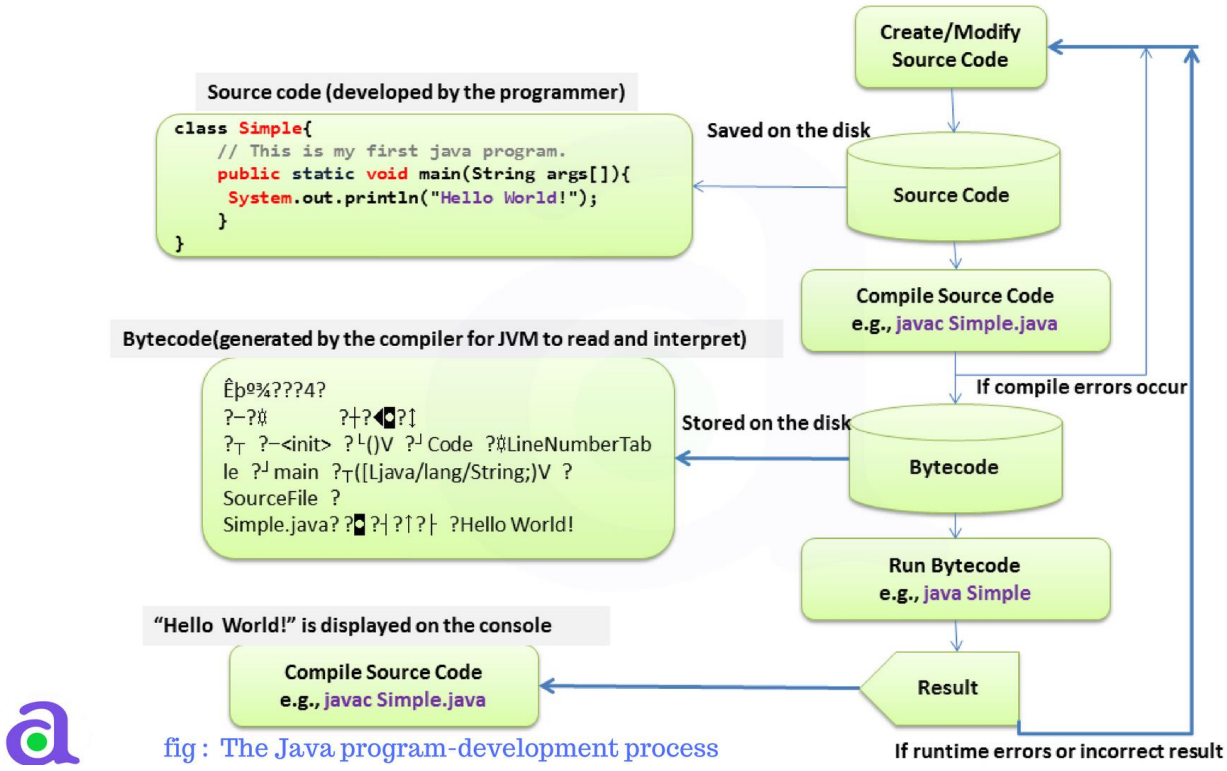


fig : The Java program-development process

4) Portable

Java language is portable as we have discussed above it facilitates the programmer the feature of carrying the Java bytecode to any platform without any implementation.

5) Secure

When it comes to security, Java is always the first choice. With java secure features it enable us to develop virus free, temper free system.

Java program always runs in Java runtime environment with almost null interaction with system OS, hence it is more secure.

- No explicit pointer
- Java Programs run inside a virtual machine sandbox
- **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

6) Robust

Robust simply means strong. Java is robust because:

- Java makes an effort to eliminate error prone codes by emphasizing mainly on compile time error checking and runtime checking.
- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

7) Architectural Neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Compiler generates bytecodes, which have nothing to do with a particular computer architecture, hence a Java program is easy to interpret on any machine.

8) Interpreted

The Java compiler generates byte-codes, rather than native machine code. To actually run a Java program, you use the Java interpreter to execute the compiled byte-codes. Java byte-codes provide an architecture-neutral object file format. The code is designed to transport programs efficiently to multiple platforms.

9) High Performance

Java is an interpreted language, so it will never be as fast as a compiled language like C or C++. But, Java enables high performance with the use of just-in-time compiler.

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code.

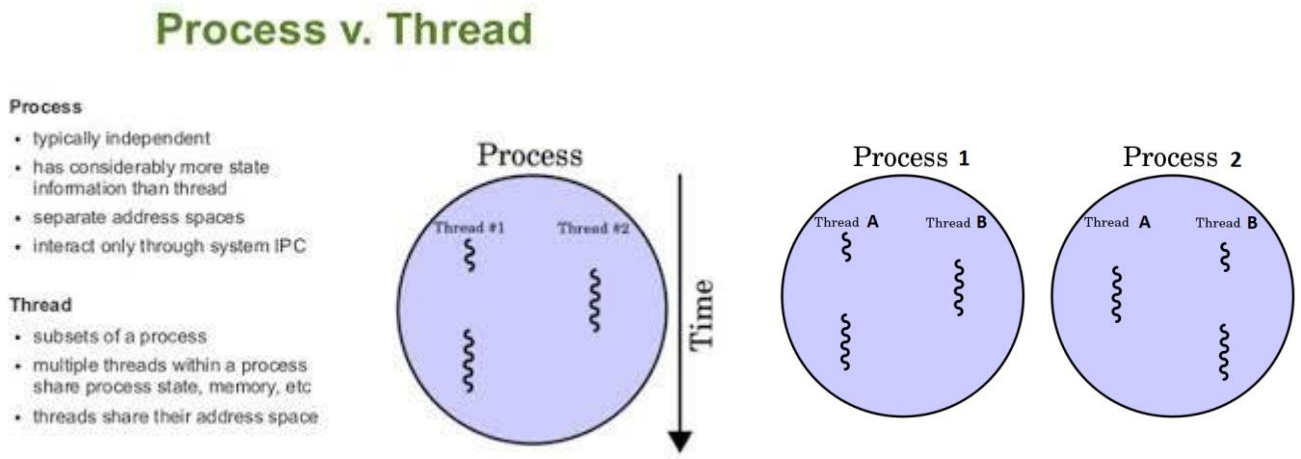
10) Multithreaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads.

The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

Java multithreading feature makes it possible to write program that can do many tasks simultaneously.

Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time, like While typing, grammatical errors are checked along.



11) Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Java is also a distributed language. Programs can be designed to run on computer networks. Java has a special class library for communicating using TCP/IP protocols. Creating network connections is very much easy in Java as compared to C/C++.

12) Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

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.

Difference between JDK, JRE, and JVM

We must understand the differences between JDK, JRE, and JVM before proceeding further to Java.

JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist**. It is a specification that provides a runtime environment in which Java bytecode can be executed**. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent.

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

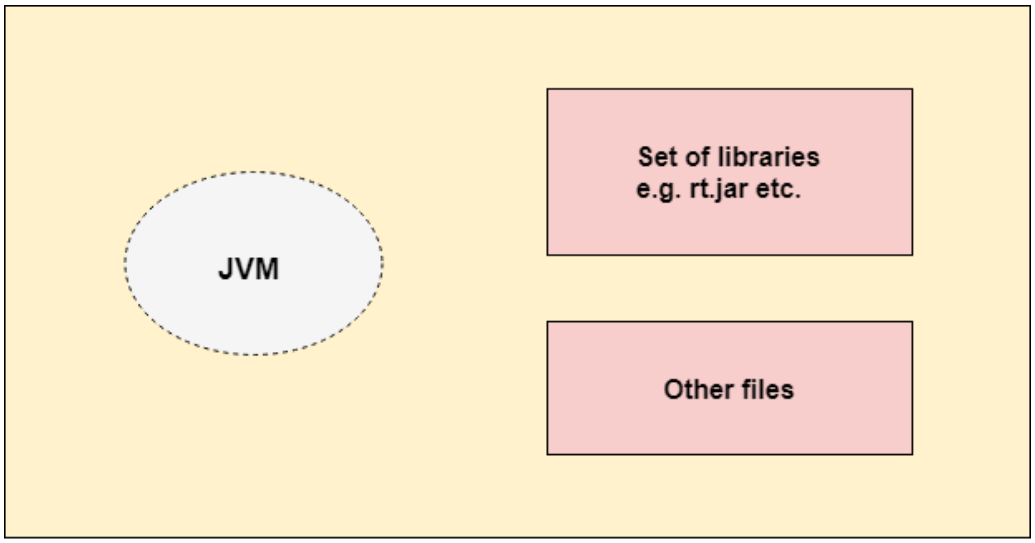
JRE

JRE stands for “Java Runtime Environment” and may also be written as “Java RTE.” The Java Runtime Environment provides the minimum requirements for executing a Java application.

It consists of the Java Virtual Machine (JVM), core classes, and supporting files.

It is:

- A **specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
- An **implementation** is a computer program that meets the requirements of the JVM specification
- **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.



JRE

JDK

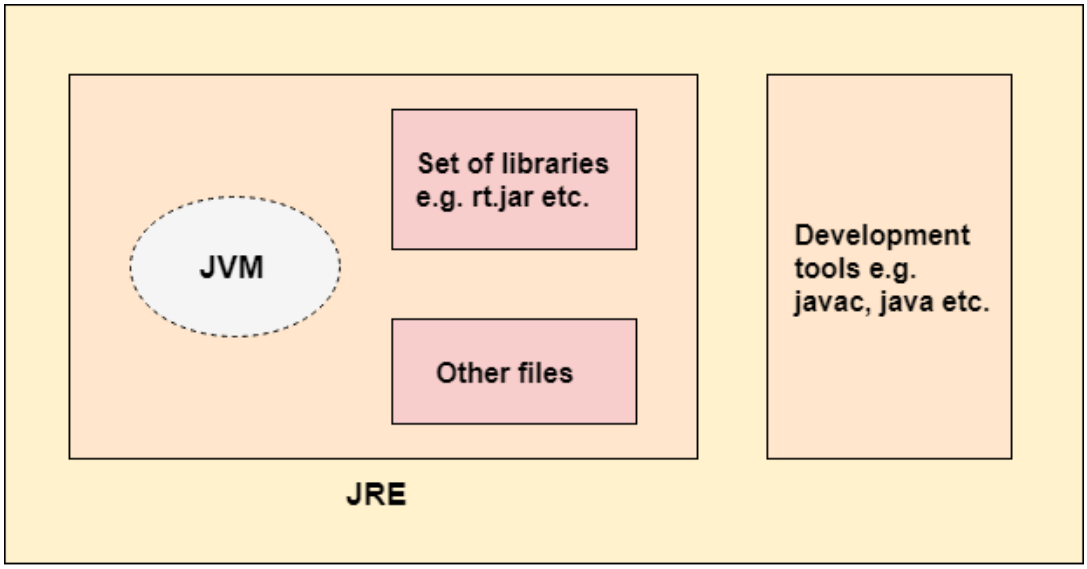
JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

Java Development Kit (in short JDK) is Kit which provides the environment to develop and execute (run) the Java program. JDK is a kit (or package) which includes two things

- Development Tools (to provide an environment to develop your java programs)
- JRE (to execute your java program).

Note: JDK is only used by Java Developers.

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



JDK

JDK = JRE + Development Tools

JRE = JVM + Library Classes

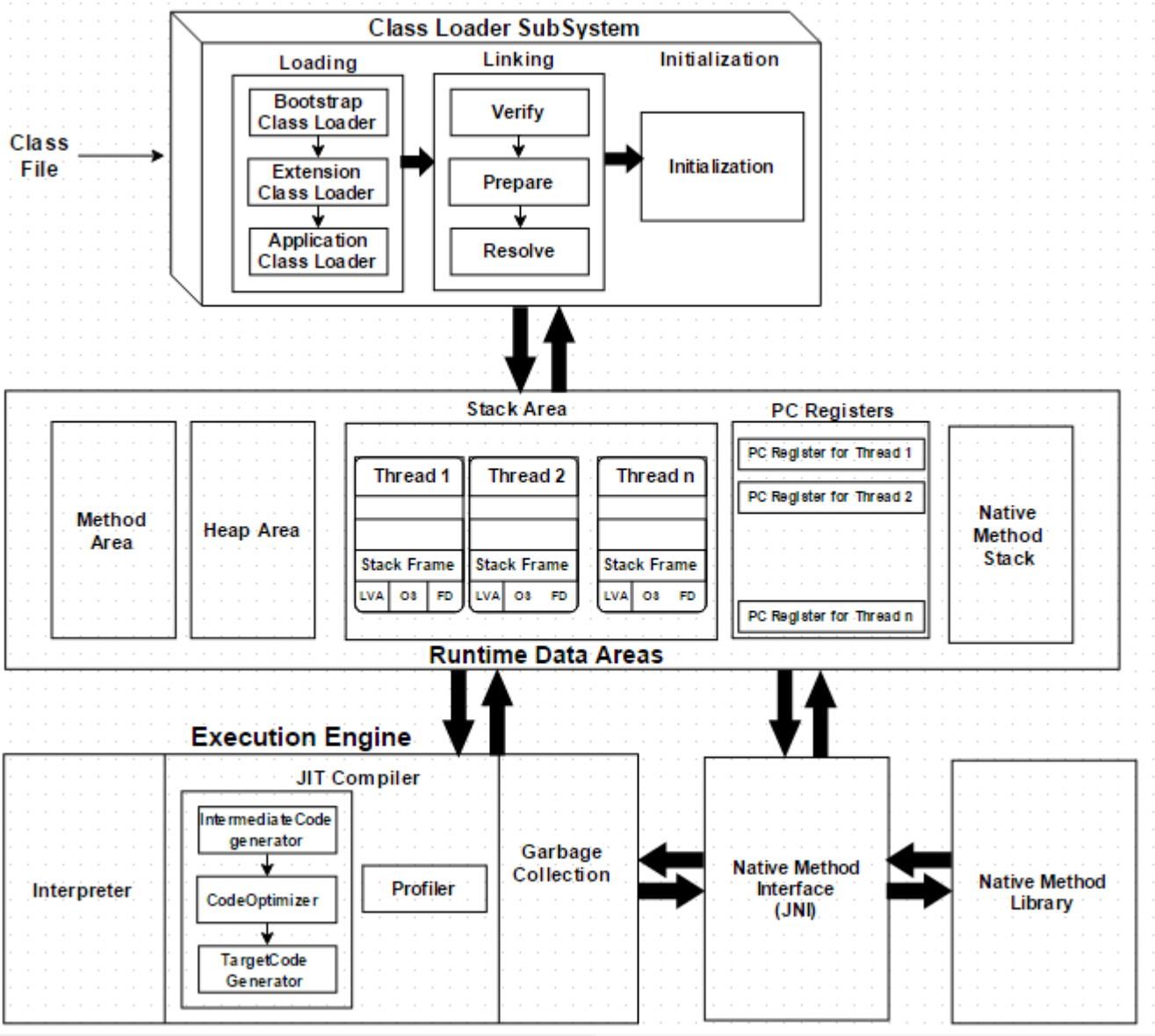
Note: JVM, JRE, and JDK are platform dependent, Java is platform independent

What is the role of JVM?

A Virtual Machine is a software implementation of a physical machine. Java was developed with the concept of WORA (Write Once Run Anywhere), which runs on a VM.

The compiler compiles the Java file into a Java .class file, then that .class file is input into the JVM, which loads and executes the class file.

Architecture of the JVM.



How Does the JVM Work?

As shown in the above architecture diagram, the JVM is divided into three main subsystems:

1. ClassLoader Subsystem
2. Runtime Data Area
3. Execution Engine

1. ClassLoader Subsystem

Java's dynamic class loading functionality is handled by the ClassLoader subsystem. It loads, links and initializes the class file when it refers to a class for the first time at runtime, not compile time.

It is mainly responsible for three activities.

- Loading
- Linking
- Initialization

1.1 Loading

Classes will be loaded by this component. BootStrap ClassLoader, Extension ClassLoader, and Application ClassLoader are the three ClassLoaders that will help in achieving it.

- **BootStrap ClassLoader** – Responsible for loading classes from the bootstrap classpath, nothing but rt.jar. Highest priority will be given to this loader. It loads core java API classes present in JAVA_HOME/jre/lib directory
- **Extension ClassLoader** – It is child of bootstrap class loader. It loads the classes present in the extensions directories JAVA_HOME/jre/lib/ext(Extension path) or any other directory specified by the java.ext.dirs system property.
- **Application ClassLoader** – It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path.

1.2 Linking

- **Verification:** It ensures the correctness of .class file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception java.lang.VerifyError.
- **Preparation:** JVM allocates memory for class variables and initializing the memory to default values.
- **Resolution:** It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

1.3 Initialization

This is the final phase of Class Loading. Here, all static variables will be assigned with the original values, and the static block will be executed.

2. Runtime Data Area

The Runtime Data Area is divided into five major components:

1) Method Area – In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

2) Heap Area – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.

3) 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 Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three sub entities:

- **Local Variable Array** – Related to the method how many local variables are involved and the corresponding values will be stored here.
- **Operand stack –** If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
- **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.

4) PC Registers – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.

5) Native Method stacks – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

3. Execution Engine

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.

1) Interpreter – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.

2) JIT Compiler – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, **but when it finds repeated code it uses the 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** – Responsible for optimizing the intermediate code generated above
3. **Target Code Generator** – Responsible for Generating Machine Code or Native Code
4. **Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.

3) Garbage Collector: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

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: This is a collection of the Native Libraries, which is required for the Execution Engine.