# 🔷 1. JDK (Java Development Kit)

**✅ Definition:**

**JDK** is a **full software development kit** used to **write, compile, and debug** Java applications. It includes everything in the JRE **plus** development tools.

**✅ Contains:**

* **JRE** (and therefore JVM)
* **Development tools**:
  + javac – Java compiler
  + java – launcher
  + javadoc – documentation tool
  + jdb – debugger
  + jar – package creator
  + jshell – interactive REPL
  + Other dev utilities and APIs

**✅ Key Point:**

* You **need JDK to develop** Java programs.

**🔁 Relationship Diagram:**

JDK = JRE + Development Tools

JRE = JVM + Libraries + Other files

**🔍 Real-World Analogy:**

* **JVM**: Like a **TV set-top box** that reads and executes channels (bytecode).
* **JRE**: Like the **TV and set-top box** combo – lets you watch programs.
* **JDK**: Like the **TV studio setup** – lets you create, record, and edit shows (applications).

# 🔷 2. JRE (Java Runtime Environment)

**✅ Definition:**

**JRE** is a **software package** that provides the environment to **run Java programs**. It contains:

* JVM
* Core Java libraries (rt.jar)
* Supporting files/configuration

**✅ Key Point:**

* JRE is **needed to run** Java applications, **not to develop** them.
* End-users or systems that just want to run a Java app install the **JRE only**.

**✅ Responsibilities:**

1. **Provides the JVM**:
   1. So that bytecode (.class files) can be executed.
2. **Provides Java Libraries (Runtime classes)**:
   1. Packages like java.lang, java.util, java.io, java.net, etc., are required **at runtime**.
   2. Without these, your program can't even do basic things like printing to console.
3. **Helps the JVM function correctly by giving it the classes it needs**

# 🔷 3. JVM (Java Virtual Machine)

**✅ Definition:**

The **JVM** is a **virtual machine** that runs Java **bytecode(.class)**. It is **platform-dependent**, meaning each OS (Windows, Linux, macOS) has a different JVM implementation.

**✅ Responsibilities:**

* Loads .class files (compiled Java bytecode)
* Verifies code for security
* Executes code (interprets or JIT compiles)
* Manages memory (heap, stack, GC)
* Provides runtime environment (**manages threads, exceptions**, etc.)

**✅ Components of JVM:**

* **Class Loader**: Loads class files
* **Bytecode Verifier**: Checks code for security issues
* **Interpreter / JIT Compiler**: Converts bytecode to machine code
* **Garbage Collector**: Reclaims memory
* **Runtime Memory Areas**:
  + Method Area
  + Heap
  + Java Stack
  + Program Counter (PC) Register
  + Native Method Stack

# Working of JVM / JVM Architecture

**JVM** is the one that calls the **main** method present in a Java code. Java applications are called **WORA (Write Once Run Anywhere)**. This means a programmer can develop Java code on one system and expect it to run on any other Java-enabled system without any adjustments. This is all possible because of the JVM. When we compile a **.java file**, **.class files** (containing byte-code) with the same class names present in the .java file are generated by the Java compiler. This **.class** file goes through various steps when we run it. These steps together describe the whole JVM.

A diagram of a computer program

AI-generated content may be incorrect.

#### Operations Performed by JVM

The various operations performed by the JVM are listed below:

1. Loading of code: The JVM loads class files that contain Java bytecode into memory. This means it finds the necessary files and gets them ready to run.
2. Verification of code: The JVM checks the bytecode it has loaded to make sure it's correct and safe.
3. Executing the code: The JVM runs the verified bytecode by turning it into machine code that the computer can understand.
4. Providing a Runtime Environment: The JVM provides a runtime environment that helps manage system resources, memory, and garbage collection.

## Core Components Of JVM

Now, we are going to discuss each component of the JVM in detail.

### 1. Class Loader Subsystem

It is mainly responsible for three activities.

#### 1. Loading:

The Class loader reads the “.*class”*file, generate the corresponding binary data and save it in the method area. For each “*.class”*file, JVM stores the following information in the method area.

* The fully qualified name of the loaded class and its immediate parent class.
* Whether the “*.class”* file is related to Class or Interface or Enum.
* Modifier, Variables and Method information etc.

After loading the “.class” file, JVM creates an object of type Class to represent this file in the heap memory. Please note that this object is of type lass predefined in **java.lang**package. These Class object can be used by the programmer for getting class level information like the name of the class, parent name, methods and variable information etc. To get this object reference we can use **getClass()** method of [Object](https://www.geeksforgeeks.org/object-class-in-java/) class.

**Example:**

// A Java program to demonstrate working  
// of a Class type object created by JVM  
// to represent .class file in memory  
import java.lang.reflect.Field;  
import java.lang.reflect.Method;  
  
// Java code to demonstrate use  
// of Class object created by JVM  
public class Geeks  
{  
 public static void main(String[] args)  
 {  
 Student s1 = new Student();  
  
 // Getting hold of Class  
 // object created by JVM.  
 Class c1 = s1.getClass();  
  
 // Printing type of object using c1.  
 System.*out*.println(c1.getName());  
  
 // getting all methods in an array  
 Method m[] = c1.getDeclaredMethods();  
 for (Method method : m)  
 System.*out*.println(method.getName());  
  
 // getting all fields in an array  
 Field f[] = c1.getDeclaredFields();  
 for (Field field : f)  
 System.*out*.println(field.getName());  
 }  
}  
  
 // A sample class whose information  
// is fetched above using its Class object.  
 class Student {  
 private String name;  
 private int roll\_No;  
  
 public String getName() { return name; }  
 public void setName(String name) { this.name = name; }  
 public int getRoll\_no() { return roll\_No; }  
 public void setRoll\_no(int roll\_no)  
 {  
 this.roll\_No = roll\_no;  
 }  
 }  
  
 Output  
 Student  
 getName  
 setName  
 getRoll\_no  
 setRoll\_no  
 name  
 roll\_No

Note: For every loaded “.class” file, only one object of the class is created.  
 Student s2 = new Student();  
 // c2 will point to same object where   
// c1 is pointing  
 Class c2 = s2.getClass();  
System.out.println(c1==c2); // true

#### 2. Linking:

Performs verification, preparation, and (optionally) resolution.

* **Verification:** It ensures the correctness of the *.class* file i.e. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*. This activity is done by the component ByteCodeVerifier. Once this activity is completed then the class file is ready for compilation.
* **Preparation:** JVM allocates memory for class static 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 the method area to locate the referenced entity.

JVM **resolves symbolic references** (like class names, method names, field names) to **actual memory addresses** (pointers).

This allows the class to be linked with other classes it depends on.

📌 Think of it like resolving com.example.MyClass → actual class object in memory.

#### 3. Initialization:

In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in the class hierarchy.

### 2. Class Loaders

There are three primary types of class loaders:

* **Bootstrap Class Loader**: It is capable of loading trusted classes. It loads core java API classes present in the **“JAVA\_HOME/lib**” directory. This path is popularly known as the bootstrap path. It is implemented in native languages like C, C++.
* **Extension Class Loader**: Loads classes from the extention JAVA\_HOME/jre/lib/ext directory or any directory specified by the java.ext.dirs system property. It is implemented in Java. It is a child of the bootstrap class loader.
* **System/Application Class Loader**: Loads classes from the application classpath, which is specified by the java.class.path environment variable. It is also implemented in Java. It is a child of the extension class loader.A diagram of a class

  AI-generated content may be incorrect.

**Example:**

// Java code to demonstrate Class Loader subsystem  
  
public class Geeks  
{  
 public static void main(String[] args)  
 {  
 // String class is loaded by bootstrap loader, and  
 // bootstrap loader is not Java object, hence null  
 System.*out*.println(String.class.getClassLoader());  
  
 // Test class is loaded by Application loader  
 System.*out*.println(Geeks.class.getClassLoader());  
 }  
}  
  
Output  
null  
jdk.internal.loader.ClassLoaders$AppClassLoader@8bcc55f

**Note:**JVM follows the **Parent Delegation Model** — each loader asks its parent first to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to the bootstrap class loader. If a class found in the boot-strap path, the class is loaded otherwise request again transfers to the extension class loader and then to the system class loader. At last, if the system class loader fails to load class, then we get run-time exception *java.lang.ClassNotFoundException*.

### 3. JVM Memory Areas

* **Method area/ class area:** In the method area, all class level information like class name, immediate parent class name, methods and variables information and Constant pool, Interfaces etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource. Method area can be of fixed or dynamic size depending on the system's configuration. Garbage collection in the method area is not mandatory for automatic memory management.

**Metaspace** 🡪 Metaspace replaced Method Area's **implementation** in Java 8. It is the memory area where the **JVM stores class metadata** (information about classes metadata(from loaded .class files), methods, fields, Runtime constant pool, annotations, Static variables (depends — if primitive/final, Method Area)) **in Java 8 and later**.

**Why Was Metaspace Introduced?**

1. **PermGen had fixed size by default**, and resizing was hard — led to frequent OutOfMemoryError: PermGen space.
2. **Metaspace is dynamically sized** (up to system memory), so more flexible.
3. Better separation of class metadata from the rest of the Java heap.

| **Java Version** | **Class Metadata Stored In** | **Memory Type** |
| --- | --- | --- |
| Java 1.1 – 7 | **Method Area** (aka PermGen) | Part of JVM Heap |
| Java 8+ | **Method Area** (implemented using Metaspace) | **Native memory** (not in Heap) |

* **Heap area:** Information of all objects is stored in the heap area and arrays, string pool(string created by literals””) are stored. There is also one Heap Area per JVM. It is also a shared resource. It is created when the JVM started. Heap can be of fixed or dynamic size depending upon the system’s configuration. Garbage collection in the heap area is mandatory for automatic memory management.

If a reference variable is declared as a **class-level field** (not inside a method), it's not in the stack — it's stored:

* In the **heap** (for instance fields),
* Or in the **method area** (for static fields)

And if it is local then reference is stored in stack and actual object in heap area.

Heap Memory is divided into several regions:

1. **Young Generation:**The area within the heap wherenew objects are generally allocated.  When this space is full, unused objects are removed by the **garbage collector**, and surviving objects move to the **Old Generation**.
2. **Old Generation:** The area within the heap where long-livedmetadata Objects are stored after surviving multiple garbage collection cycles. Holds **older objects** that are no longer frequently used. These objects stay in memory unless explicitly removed.

Starting with Java 8, the old permanent generation space was removed and replaced by metaspace. Metaspace uses native memory outside the main heap. This change helps Java to handle class metadata better and also reduces the chances of errors caused by permanent generation.

* **Stack area:** For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.

The main components of a stack frame are listed below:

* + Method call information: Stores details about the method being executed.
  + Local variables: Stores local variables defined within the method.
  + Method parameters: Stores parameters passed to the method.
  + Return address: Tracks where to return once the method is completed.

The stack frame basically consists of three parts. The three parts are

* **Local Variable Array**
* **Operand Stack -** The JVM uses an operand stack as a workspace for temporary calculations.
* **Frame Data –** 
  1. It contains all symbolic references (constant pool resolution) and normal method returns related to that particular method.
  2. It also contains a reference to the Exception table which provides the corresponding catch block information in the case of exceptions.
* **PC Registers:** Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
* **Native method stacks:** For every thread, a separate native stack is created. It stores native method information. Native method stacks handle the execution of native methods that interact with the Java code.

A diagram of a data processing process

AI-generated content may be incorrect.

**Memory Layout of an Object**

In Java, an object consist of three main parts:

* **Object header:** It contains metadata like the object's class and lock information for synchronization.
* **Instance data**: It refers to the actual fields or variables that belongs to an object.
* **Padding**: It refers to the extra memory to align the object size to the word size for better performance.

The **object header** includes:

* **Class Pointer:** This is a reference to the object's class.
* **Lock Information**: This is used for synchronization in multithreaded programs.
* **GC-related Information**: It is used by the garbage collector for managing the object lifecycle.

**Constant Pool**

The **Constant Pool** (a.k.a. **Runtime Constant Pool**) is a part of the **class metadata** where the JVM stores **constants** referenced by a class, such as:

* String literals ("hello")
* Final primitive constants (static final int MAX = 100)
* Method names
* Field names
* Class/interface references
* Type descriptors

| **Type of Constant Pool** | **Location** | **Description** |
| --- | --- | --- |
| **Class-level Constant Pool** | 🟩 Method Area (per class) | Stores symbolic references & literals used by the class |
| **String Constant Pool** | 🟩 Heap (special string pool area) | Stores interned string literals ("abc", "xyz") |

**📌 Summary**

| **Constant Type** | **Where Stored** | **Memory Area** |
| --- | --- | --- |
| Class constants | Runtime Constant Pool | Method Area |
| String literals | String Pool | Heap (special area) |
| Final primitives | Constant Pool | Method Area |
| Object references | Regular Heap or Stack | Heap + Stack |

### 4. Execution Engine

Execution engine executes the “.class” (bytecode). It reads the byte-code line by line, uses data and information present in various memory areas and executes instructions. It can be classified into three parts:

* **Interpreter:** It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
* **Just-In-Time Compiler(JIT):** It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.
* **Garbage Collector:** It destroys unreferenced objects. For more on Garbage Collector, refer [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/).

### 5. Java Native Interface (JNI)

It is an interface that interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

### 6. Native Method Libraries

These are collections of native libraries required for executing native methods. They include libraries written in languages like C and C++.

**🔚 Summary Table:**

| **Feature** | **JVM** | **JRE** | **JDK** |
| --- | --- | --- | --- |
| Runs Java Bytecode | ✅ Yes | ✅ Yes | ✅ Yes |
| Contains JVM | ✅ Yes | ✅ Yes | ✅ Yes |
| Contains Libraries | ❌ No | ✅ Yes | ✅ Yes |
| Contains Compiler | ❌ No | ❌ No | ✅ Yes (javac) |
| For Development | ❌ No | ❌ No | ✅ Yes |
| For Running Apps | ✅ Yes | ✅ Yes | ✅ Yes |

# 🔷**🧠 What "Loading a Class" Means:**

* "Loading" a class means **reading its .class bytecode** into memory so that the JVM can **use and execute** it.
* The ClassLoader does this automatically as needed — **lazy loading**.

### Why is ClassLoader Needed?

Because Java is **modular and dynamic** — it doesn’t load all classes at once.

It only loads what is needed, when it’s needed (lazy loading).

This keeps memory usage low and startup fast.

### Paths 🡪

1. **Classpath** - The **classpath** is a **parameter** that tells the JVM **where to look** for user-defined .class files, JAR files, or packages when running Java programs.

* **Temporarily via command line**:

java -cp .;lib/mylib.jar MyApp # Windows

java -cp .:lib/mylib.jar MyApp # Linux/macOS

* **Permanently via environment variable**:

export CLASSPATH=.:lib/mylib.jar # Linux/macOS

set CLASSPATH=.;lib\mylib.jar

1. SourcePath - Used during **compilation** to tell javac where to find .java source files.

javac -sourcepath src -d out src/com/example/HelloWorld.java

1. **Module Path (Java 9+) -** With the introduction of Java Modules (JPMS), we now also have:

java --module-path mods -m my.module/com.example.Main

1. **Path (java.library.path) -** Used to tell JVM where to find **native libraries** (.dll, .so, .dylib)

java -Djava.library.path=/usr/lib myapp

**🔁 Comparison Table**

| **Path Type** | **Used By** | **Used For** | **Example** |
| --- | --- | --- | --- |
| **Classpath** | JVM & javac | Finding .class files, JARs | -cp .:lib/mylib.jar |
| **Sourcepath** | javac | Finding .java source files | -sourcepath src/ |
| **Module Path** | JVM & javac | Finding Java 9+ modules | --module-path mods/ |
| **Library Path** | JVM | Finding native (C/C++) libraries |  |

**Can the same .class file be loaded multiple times by different class loaders?**

✅ **Answer:**

Yes — the **same .class file can be loaded multiple times by different class loaders**.  
Each class loader creates a **separate namespace**, so even if it loads the same .class, it results in **different Class objects** in memory.

**🔹 1. How JVM Identifies a Class**

The JVM **does not** identify a class **just by its name**.  
It uses this combination: <fully-qualified-class-name, ClassLoader instance>

So the same class name (e.g., com.example.MyService) loaded by **two different class loaders** is treated as **two completely different classes** in the JVM.

**🔍 2. What Happens Internally?**

Even though both class loader loads the **same .class file**, the two Class<?> objects (c1 and c2) are **not equal** and **not interchangeable** — JVM treats them as **two unrelated classes**.

**🔄 3. Why is this useful?**

* It allows **modular class loading**.
* Enables **hot deployment/redeployment** in servers like Tomcat without restarting the JVM.
* Allows **isolation** in plugin frameworks (e.g., Eclipse, OSGi).
* Supports **multiple versions** of the same class/library at runtime.

**🔥 Real-World Examples**

**🔸 Tomcat / Spring Boot Dev Tools**

* When you redeploy a web app, Tomcat uses a new ClassLoader for the new version of your classes.
* Old classes become **unreachable** and eligible for GC, but still **exist in memory**.

**🔸 Spring Boot DevTools**

* Uses a **RestartClassLoader** to isolate application code from framework code.
* This enables **automatic restart** of only your app classes while reusing Spring itself.

**🔸 OSGi / Eclipse**

* OSGi bundles are isolated — each has its own ClassLoader.
* You can have com.example.Foo in multiple bundles with **different versions** — each loaded independently.

**❗ Risks and Gotchas**

* **ClassCastException** can occur:

if (c1 != c2) {

MyClass obj = (MyClass)c2.newInstance(); // ClassCastException if expecting c1 type

}

* **Memory leaks** in app servers (like Tomcat):
  + If you store references to classloader-loaded objects in static fields, you prevent class unloading during redeployment.