

JAVA NOTES

GLOBAL INSTITUTE OF TECHNOLOGY AND MANAGEMENT

UNIT 1

DESIGNED BY

java CSE 4th Sem

Assistant Professor

Introduction to Java

Muzamil Aslam

What is Java?

What is Java?

- Java is an Object-Oriented Programming Language.
- It was developed by James Gosling and first released by Sun Microsystems in 1995.
- More than 3 billion devices run Java programs.



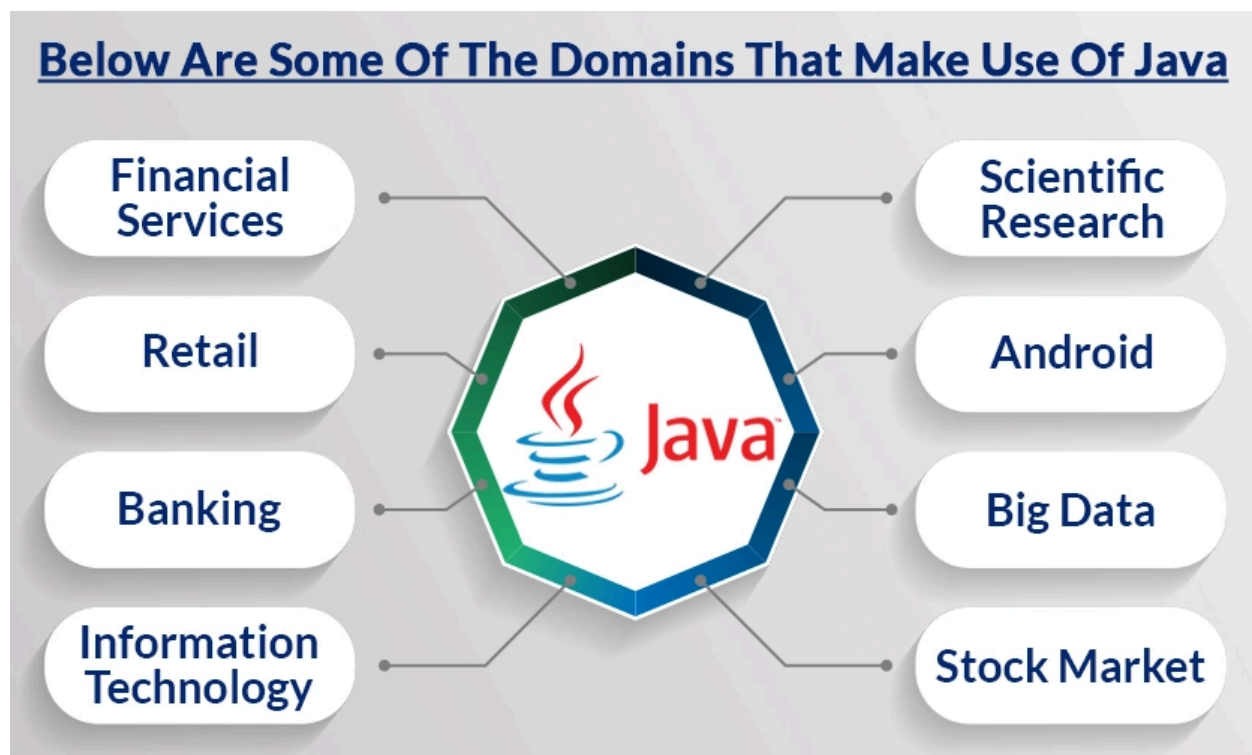
Java is a popular programming language, created in 1995.

It is owned by Oracle, and more than **3 billion** devices run Java.

It is used for:

- Mobile applications (specially Android apps)
- Desktop applications
- Web applications
- Web servers and application servers
- Games
- Database connection
- And much, much more!

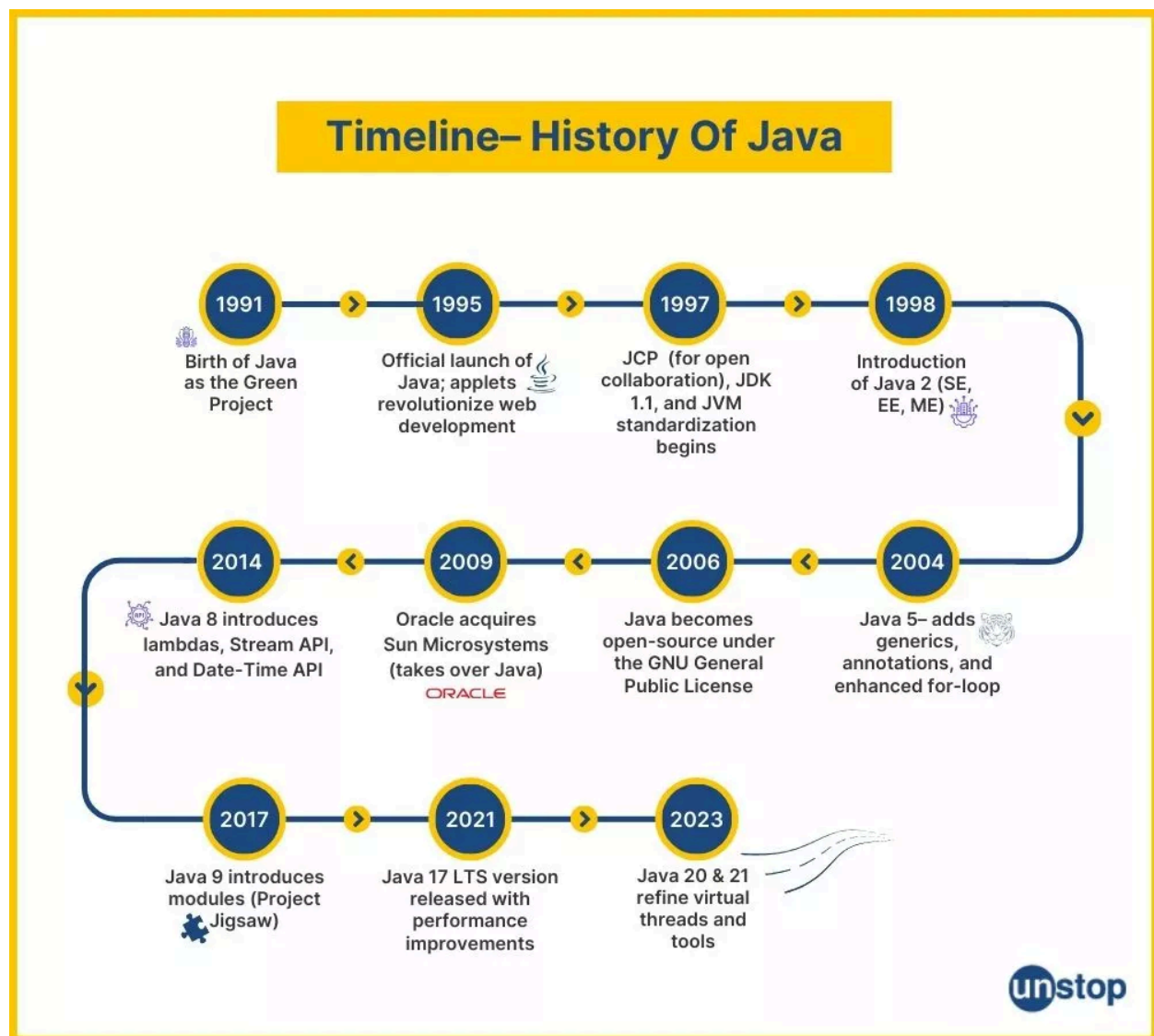
Why Use Java?




- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming languages in the world
- It has a large demand in the current job market
- It is easy to learn and simple to use
- It is open-source and free

- It is secure, fast and powerful
- It has huge community support (tens of millions of developers)
- Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs
- As Java is close to [C++](#) and [C#](#), it makes it easy for programmers to switch to Java or vice versa

The Complete History of Java Programming Language





[Java](#) is an [Object-Oriented programming](#) language developed by **James Gosling** in the early 1990s. The team initiated this project to develop a language for digital devices such as set-top boxes, television, etc. Originally [C++](#) was considered to be used in the project but the idea was rejected for several reasons (For instance C++ required more memory). Gosling endeavoured to alter and expand [C++](#) however before long surrendered that for making another stage called **Green**. James Gosling and his team called their project “**Greentalk**” and its file extension was **.gt** and later became to known as “**OAK**”.

Why “Oak”?

The name **Oak** was used by **Gosling** after an **oak tree** that remained outside his office. Also, Oak is an image of solidarity and picked as a national tree of numerous nations like the U.S.A., France, Germany, Romania, etc. But they had to later rename it as “**JAVA**” as it was already a trademark by **Oak Technologies**. “**JAVA**” Gosling and his team did a brainstorm session and after the session, they came up with several names such as **JAVA, DNA, SILK, RUBY, etc.** **Java** name was decided after much discussion since it was so unique.

The name Java originates from a sort of **espresso bean**, Java. Gosling came up with this name while having a coffee near his office. Java was created on the principles like **Robust, Portable, Platform Independent, High Performance, Multithread, etc.** and was called one of the **Ten Best Products of 1995** by the **TIME MAGAZINE**. Currently, Java is used in **internet programming, mobile devices, games, e-business solutions, etc.**

The [Java language](#) has experienced a few changes since **JDK 1.0** just as various augmentations of classes and packages to the standard library. In Addition to the language changes, considerably more sensational changes have been made to the Java Class Library throughout the years, which has developed from a couple of hundred classes in JDK 1.0 to more than three thousand in J2SE 5.

1. Early Development (1991-1995)

- **1991:** James Gosling and his team at **Sun Microsystems** started working on a project called *Oak* (later renamed Java).
- **1995:** Java 1.0 was released with the "**Write Once, Run Anywhere**" principle.

2. Major Java Versions and Features

Version	Year	Key Features
JDK 1.0	1995	First release, Applets, AWT (Abstract Window Toolkit).
JDK 1.1	1997	Inner classes, JDBC (Java Database Connectivity), JavaBeans.
J2SE 1.2	1998	Collections framework, Swing GUI, Just-In-Time (JIT) compiler.
J2SE 1.3	2000	HotSpot JVM, JavaSound API.
J2SE 1.4	2002	Assertion, regular expressions, NIO (New Input/Output).
J2SE 5.0	2004	Generics, Enhanced for-loop, Auto-boxing, Enum, Varargs.
Java SE 6	2006	Scripting API, Web Services, Performance improvements.
Java SE 7	2011	Try-with-resources, Diamond Operator (<>), NIO.2.
Java SE 8	2014	Lambda Expressions, Stream API, Functional Interfaces, Default Methods.
Java SE 9	2017	Module System (Project Jigsaw), JShell (REPL).
Java SE 10	2018	var keyword (local variable type inference).
Java SE 11 (LTS)	2018	Long-Term Support (LTS), HTTP Client API.
Java SE 12-16	2019-2021	Switch Expressions, Pattern Matching, Records.
Java SE 17 (LTS)	2021	Sealed Classes, Foreign Function & Memory API.
Java SE 21 (LTS)	2023	Virtual Threads, Structured Concurrency.

3. Key Milestones in Java Evolution

1995 – Java 1.0

- **First version released** with Applets and AWT.
- "Write Once, Run Anywhere" using **Java Virtual Machine (JVM)**.

2004 – Java 5 (J2SE 5.0)

- **Major enhancements:** Generics, Auto-boxing, Enhanced for-loop (`for-each`).

2014 – Java 8

- **Lambda Expressions** and **Streams API** revolutionized functional programming in Java.

2017 – Java 9

- **Modularization (Project Jigsaw)** helped in structuring large applications.

2018 – Java 11 (LTS)

- **Long-Term Support (LTS)** version introduced new API enhancements.

2023 – Java 21 (LTS)

- **Virtual Threads** improved performance for concurrent applications

2. Object-Oriented Programming (OOP) Structure in Java



Java follows the **Object-Oriented Programming (OOP) paradigm**, which makes code reusable, modular, and scalable. The main **OOP principles** in Java include:

1. Classes and Objects

- **Class:** A blueprint for creating objects.
- **Object:** An instance of a class with states (fields) and behaviors (methods).

2. Encapsulation

- **Definition:** Wrapping data (variables) and code (methods) together.
- **Achieved using:** Private variables and public getter/setter methods

3. Inheritance

- **Definition:** Acquiring properties and behavior from another class.
- **Uses `extends` keyword.**

4. Polymorphism

- **Definition:** Ability of a method to behave differently based on the object.
- **Two types:**
 - **Method Overloading** (Compile-time Polymorphism)
 -

5. Abstraction

- **Definition:** Hiding implementation details and showing only necessary parts.
- **Achieved using:**
 - **Abstract classes** (`abstract` keyword)
 - **Interfaces** (`interface` keyword)

6. Interfaces

- **Definition:** A contract that a class must follow.
- **Uses `interface` keyword.**
- **Supports multiple inheritance.**

Java Editions

Java comes in different editions, each serving specific purposes:

1. **Java Standard Edition (Java SE)**
 - Used for developing desktop and server-side applications.

- o Includes **core libraries** like Collections, Multithreading, etc.
- 2. **Java Enterprise Edition (Java EE)**
 - o Used for **web and enterprise applications** (Servlets, JSP, EJB).
- 3. **Java Micro Edition (Java ME)**
 - o Designed for **embedded systems, mobile applications**.
- 4. **JavaFX**
 - o Used for **modern GUI applications**.

Overview and Characteristics of Java

Overview of Java

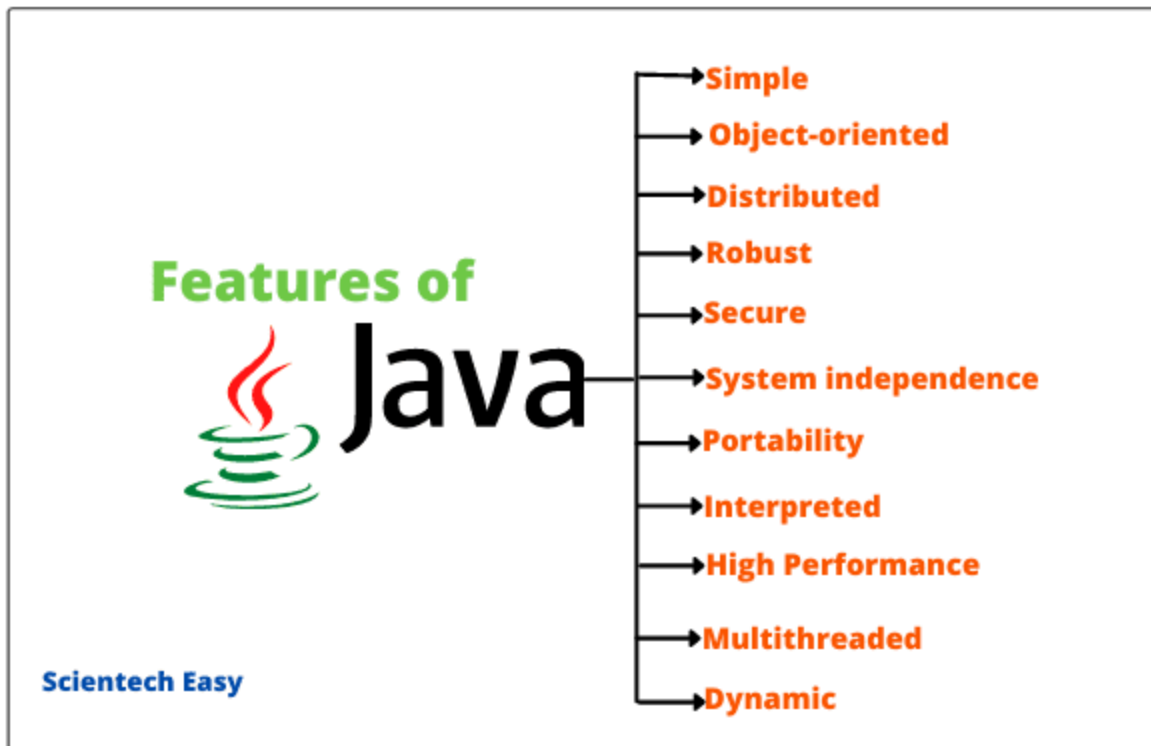
Java is a **high-level, object-oriented, and platform-independent** programming language developed by **James Gosling** at **Sun Microsystems** (now owned by Oracle) and released in **1995**. It is widely used for **web applications, enterprise applications, mobile development (Android), and cloud computing**.

Java follows the "**Write Once, Run Anywhere**" (**WORA**) principle, meaning Java code can run on any platform that has a **Java Virtual Machine (JVM)**.

Key Components of Java

1. **Java Development Kit (JDK)** – Includes Java compiler (`javac`), libraries, and tools to develop Java applications.
 2. **Java Runtime Environment (JRE)** – Contains the necessary runtime libraries and JVM to execute Java applications.
 3. **Java Virtual Machine (JVM)** – Converts Java **bytecode** into machine-specific code for execution.
-

Characteristics of Java



Java has several features that make it powerful, secure, and versatile.

1. Platform Independence

- Java programs are compiled into **bytecode**, which runs on any operating system with a **JVM**.
- No need to recompile for different platforms.

2. Object-Oriented

- Java follows **OOP principles**:
 - **Encapsulation** (data hiding using private variables)
 - **Inheritance** (reusing properties and methods)
 - **Polymorphism** (method overloading and overriding)
 - **Abstraction** (hiding implementation details)

3. Simple and Easy to Learn

- Java syntax is similar to **C++**, but it eliminates complex features like **pointers and multiple inheritance**.

4. Secure

- No direct memory access (no pointers).

- Features like **bytecode verification** and **Security Manager** protect applications from vulnerabilities.

5. Robust

- Strong **exception handling** using `try-catch-finally`.
- **Garbage collection** automatically manages memory.

6. Multi-threaded

- Supports **multi-threading**, allowing parallel execution of tasks.
- Uses `Thread` class and `Runnable` interface.

7. High Performance

- **Just-In-Time (JIT) Compiler** optimizes code execution.
- Runs faster than interpreted languages like Python but slightly slower than C++.

8. Distributed Computing

- Java supports **networking** and **remote method invocation (RMI)** for distributed applications.

9. Dynamic and Extensible

- Supports **dynamic class loading**, meaning classes are loaded when required.
- Integrates with various **APIs, libraries, and frameworks**.

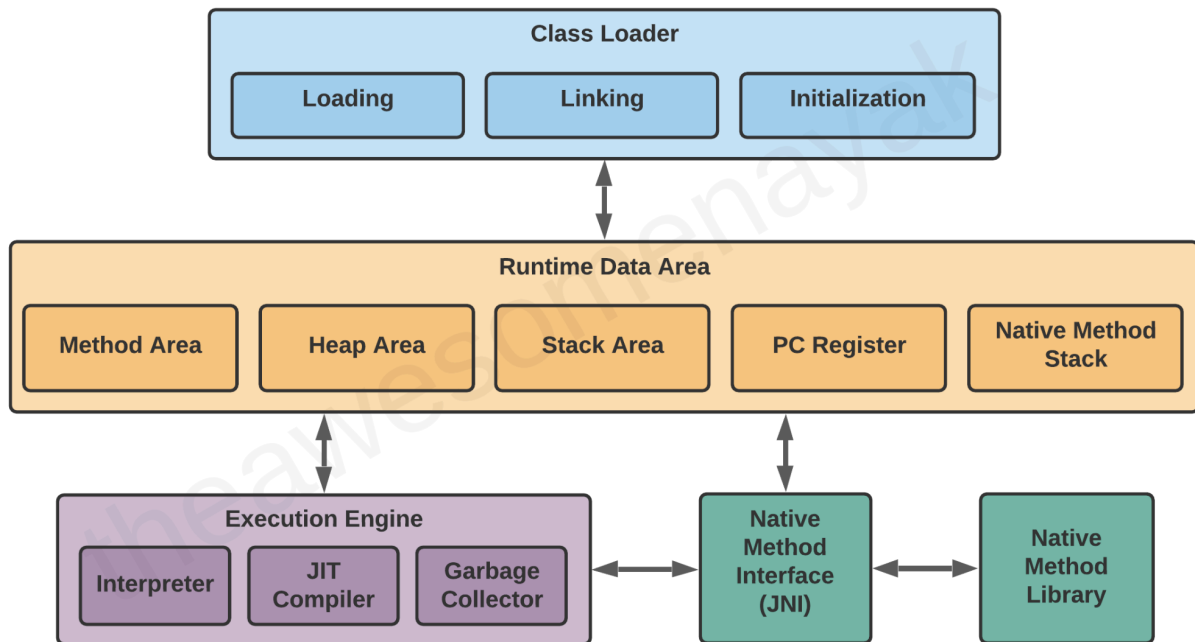
10. Memory Management (Automatic Garbage Collection)

- Java automatically reclaims unused memory through **Garbage Collection**.
- No need for manual memory allocation like in C/C++.

Organization of the Java Virtual Machine (JVM)

The **Java Virtual Machine (JVM)** is the core component of Java's platform independence, responsible for executing Java programs by converting **bytecode** into machine code. The JVM acts as a runtime engine that provides security, memory management, and execution of Java applications.

1. Components of JVM



1.1 Class Loader Subsystem

- **Loads Java classes** dynamically at runtime.
- **Steps in Class Loading:**
 1. **Loading** – Loads `.class` files into memory.
 2. **Linking** – Verifies, prepares, and resolves dependencies.
 3. **Initialization** – Static variables and blocks are initialized.

1.2 Runtime Data Areas (JVM Memory Structure)

JVM memory is divided into different sections:

1.2.1 Method Area (Class Area)

- Stores **class metadata**, static variables, and method information.
- **Shared among all threads.**

1.2.2 Heap Area

- Stores **objects and instance variables.**
- **Managed by Garbage Collector.**

1.2.3 Stack Area

- Stores method **execution frames** for each thread.
- Each method call creates a **new stack frame** (local variables, operand stack).

1.2.4 PC (Program Counter) Register

- Holds the **address of the next instruction** to be executed.
- Each thread has its own **PC register**.

1.2.5 Native Method Stack

- Used for executing **native (non-Java) methods**, like C/C++ libraries.

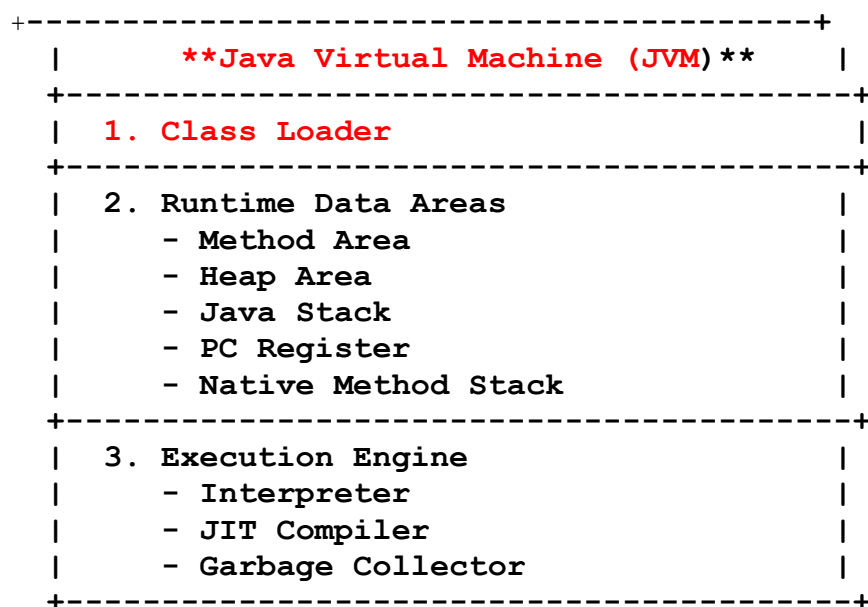
2. Execution Engine

- Converts **bytecode into machine code**.
- **Key components:**
 1. **Interpreter** – Executes bytecode **line by line** (slow).
 2. **Just-In-Time (JIT) Compiler** – Converts bytecode into **native machine code** for faster execution.
 3. **Garbage Collector (GC)** – Reclaims unused memory in the Heap.

3. Native Interface

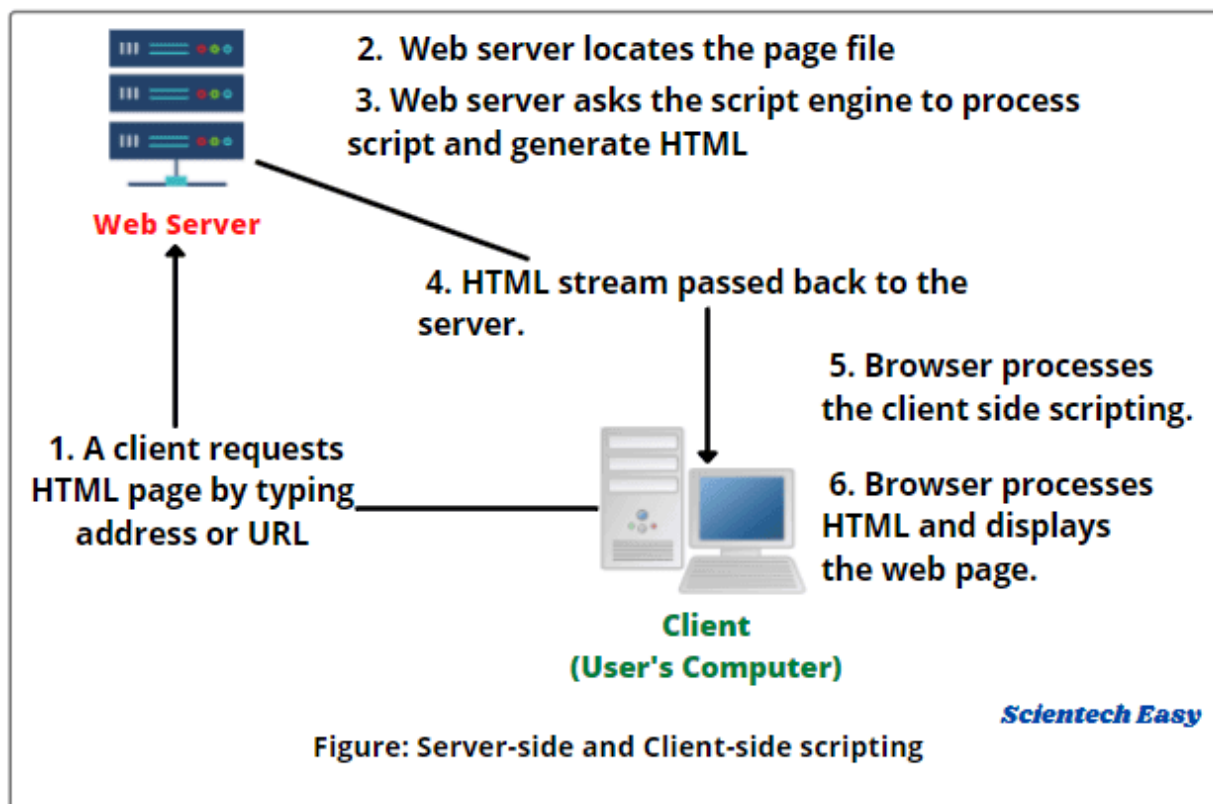
- **Java Native Interface (JNI)** allows Java to call **C/C++ native libraries**.
- **Native Method Libraries:** Provide OS-specific functionality.

JVM Architecture Diagram



Client-Side Programming, Platform Independence & Portability in Java

1. Client-Side Programming in Java



Client-side programming refers to code that runs on the user's device (client) rather than on a server. In Java, client-side programming is commonly used for **Graphical User Interfaces (GUIs)**, **Web Applications**, and **Desktop Applications**.

Client-Side Technologies in Java

1. **Java Applets (Deprecated)**
 - Used to run Java applications in web browsers (no longer supported in modern browsers).
2. **Swing (GUI Toolkit)**

- Used for developing **desktop applications** (e.g., calculators, text editors).
- Example:

```
import javax.swing.*;
public class SimpleGUI {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Client-Side GUI");
        JButton button = new JButton("Click Me!");
        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

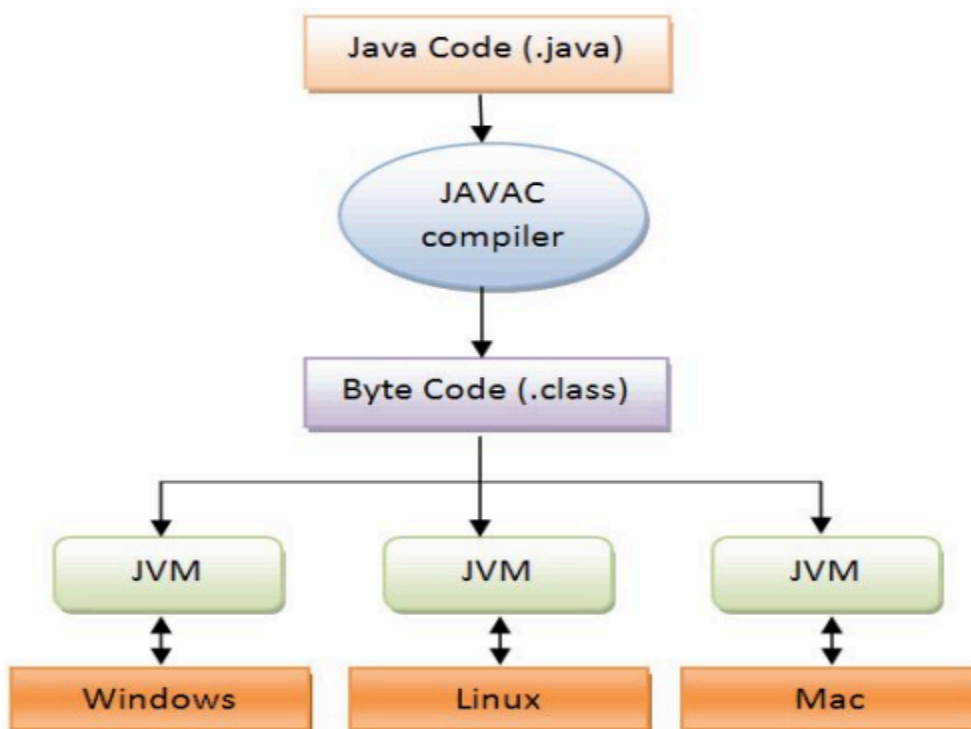
3. JavaFX (Modern GUI Framework)

- Used for **rich internet applications (RIA)** with **CSS and XML**.

4. Servlets and JSP (Java Server Pages)

- Used for **web-based client applications**.

2. Platform Independence in Java



What is Platform Independence?

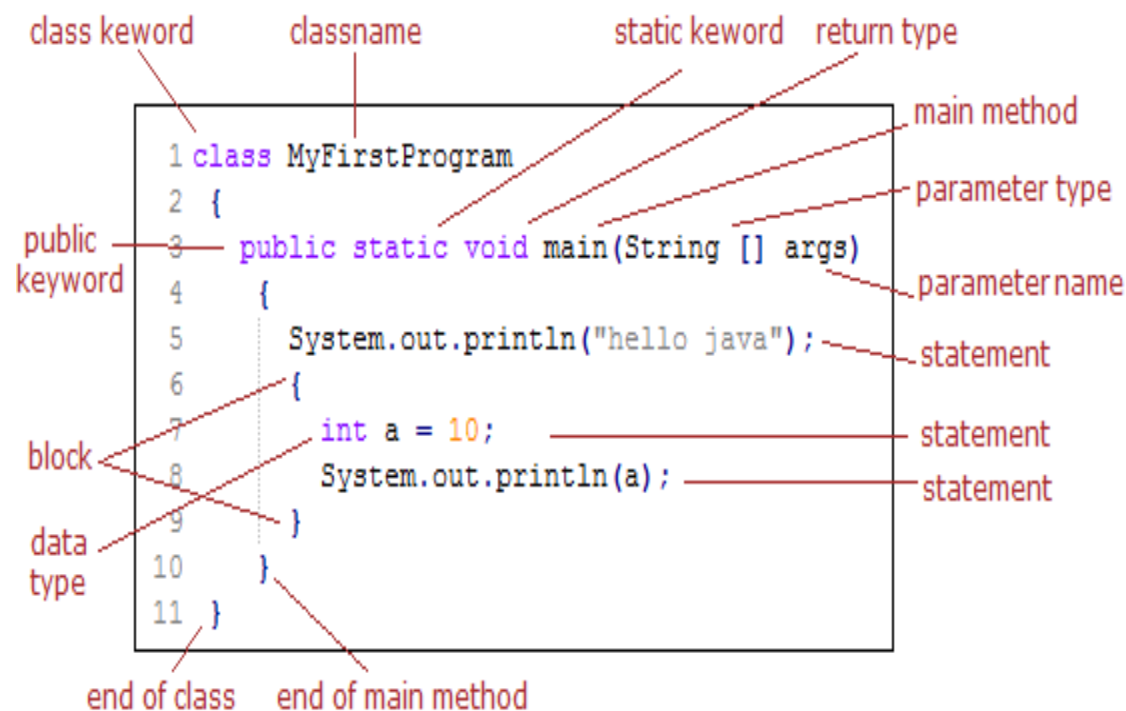
- A Java program can run on any operating system (Windows, Linux, macOS) without modification.
- This is possible because Java code is compiled into **bytecode** (.class file), which runs on the **Java Virtual Machine (JVM)**.

How Platform Independence Works?

1. **Java Source Code (.java)** → Compiled using `javac`.
2. **Bytecode (.class)** → Runs on any OS using **JVM**.
3. JVM converts **bytecode** into **machine-specific code**.

Example

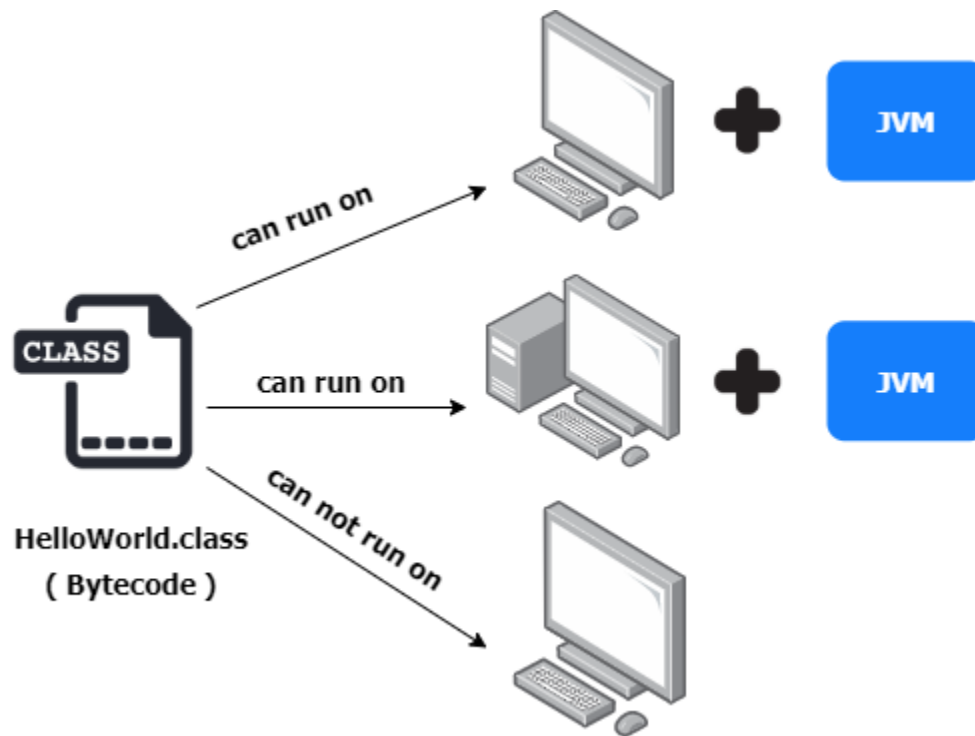
If you write a Java program:



```
public class Example {
    public static void main(String[] args) {
        System.out.println("Platform Independent Java!");
    }
}
```

- **Compile:** `javac Example.java` (Generates `Example.class`)
- **Run on any OS:** `java Example`

3. Portability in Java



What is Portability?

- Java programs can be executed on different systems without modification.
- Java achieves portability through **JVM** and **Java APIs**.

Reasons for Java's Portability

1. **Bytecode Execution:** Runs on any device with a JVM.
2. **Standard Libraries (Java API):** Same code works across different platforms.
3. **No Hardware Dependency:** Unlike C/C++, Java does not require recompilation for different systems.

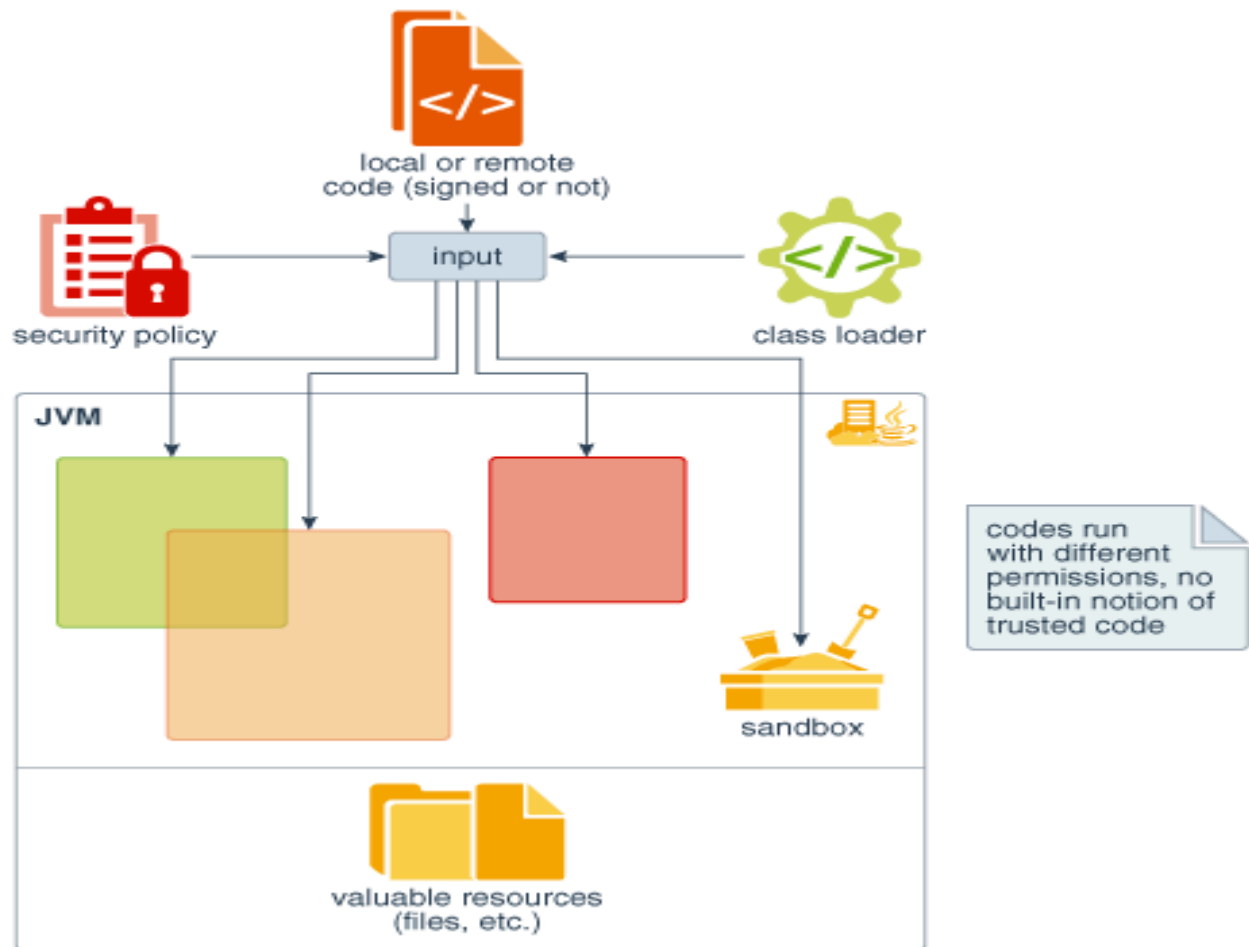
Example of Portability

- A Java program compiled on **Windows** can be executed on **Linux** without any changes.

Java Security: Security Architecture, Policy, and Sandbox Model

Java provides a **robust security model** to protect applications from unauthorized access, malicious code execution, and data breaches. Java's security mechanism is based on **encryption, authentication, access control, and sandboxing**.

1. Java Security Architecture



The **Java Security Architecture** is designed to ensure **safe execution of Java applications** by implementing **security policies, authentication, and runtime checks**.

Key Components of Java Security Architecture

1. Class Loader

- Ensures only trusted and verified classes are loaded into the JVM.
- Prevents **unauthorized code execution**.

2. Bytecode Verifier

- Checks Java bytecode before execution.

- Prevents **malformed or unauthorized code** from running.
 - 3. **Security Manager**
 - Enforces security policies at runtime.
 - Prevents applications from accessing sensitive resources.
 - 4. **Access Control Mechanism**
 - Uses **Java Permissions API** to restrict file, network, and system access.
 - Implemented via **security policies**.
 - 5. **Cryptography & Encryption**
 - Java provides APIs for **encryption, hashing, and digital signatures** (`java.security` and `javax.crypto`).
 - 6. **Authentication & Authorization**
 - Supports **Java Authentication and Authorization Service (JAAS)** for user authentication.
-

2. Java Security Policy

Java uses a **policy-based security model**, allowing administrators to define **access control rules**.

Security Policy Features

- **Security policies** are stored in a **policy file** (`.policy` file).
- Policies specify **permissions** for Java applications.
- Controlled using **SecurityManager** and **AccessController**.

Example: Java Policy File

```
grant codeBase "file:/myApp/" {  
    permission java.io.FilePermission "/data/*", "read, write";  
    permission java.net.SocketPermission "localhost:8080", "connect, accept";  
};
```

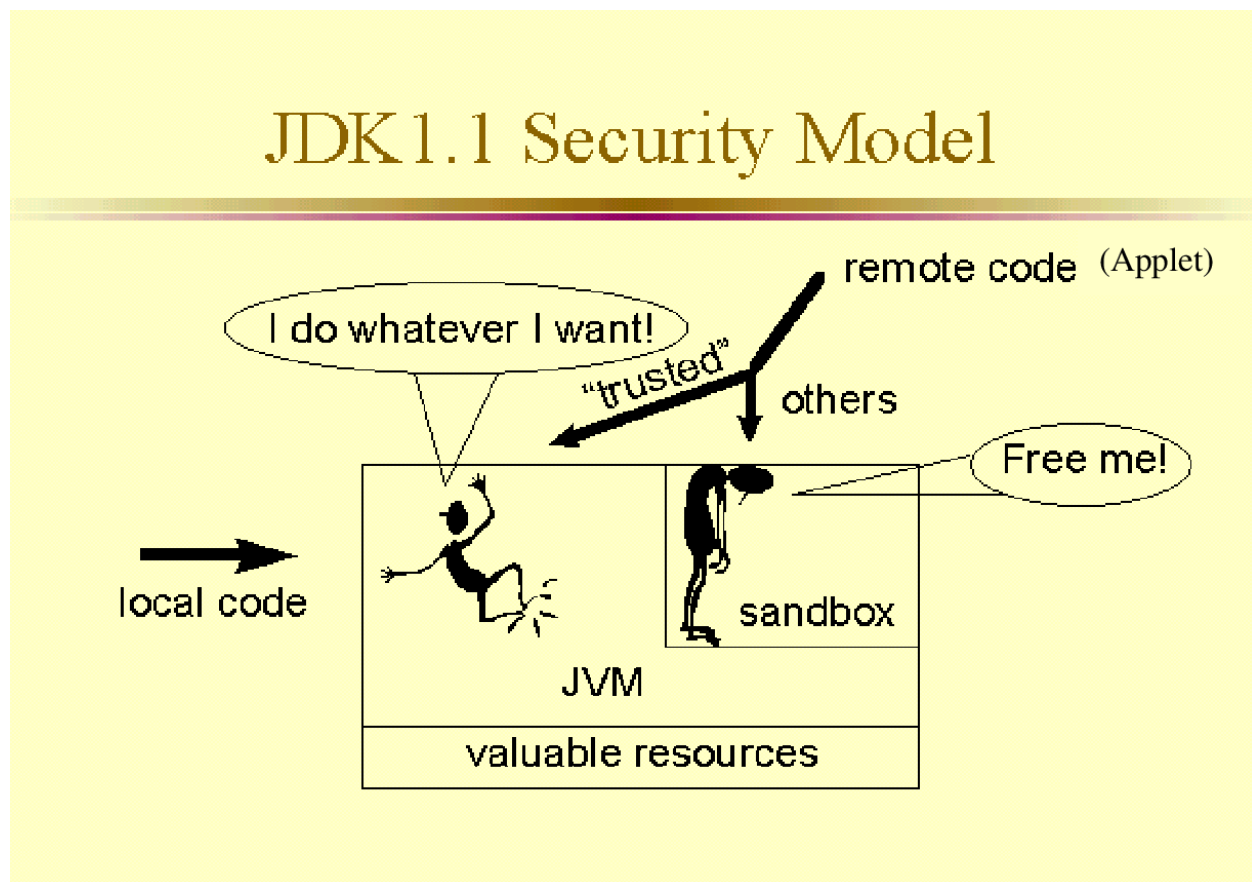
- Grants file read/write permissions for `/data/`.
- Grants **network connection** permission to `localhost:8080`.

Enforcing Security Policy in Java

```
public class SecurityTest {  
    public static void main(String[] args) {  
        SecurityManager sm = System.getSecurityManager();  
        if (sm == null) {  
            System.setSecurityManager(new SecurityManager());  
        }  
        System.out.println("Security Manager is enabled.");  
    }  
}
```

- **SecurityManager** enforces Java security policies at runtime.

3. Java Sandbox Model



The **Sandbox Model** is a security mechanism that restricts untrusted Java code (such as applets) from performing harmful operations.

How the Java Sandbox Works

1. **ClassLoader** loads code into a **restricted environment**.
2. **Bytecode Verifier** ensures code follows Java security rules.
3. **Security Manager** applies security policies.
4. **Access Control** restricts file, network, and system operations.

Types of Java Code in the Sandbox

- **Trusted Code** (Full Access): Locally installed Java applications with unrestricted permissions.
- **Untrusted Code** (Restricted): Applets or downloaded code running in the sandbox.

Example: Java Security with Sandbox

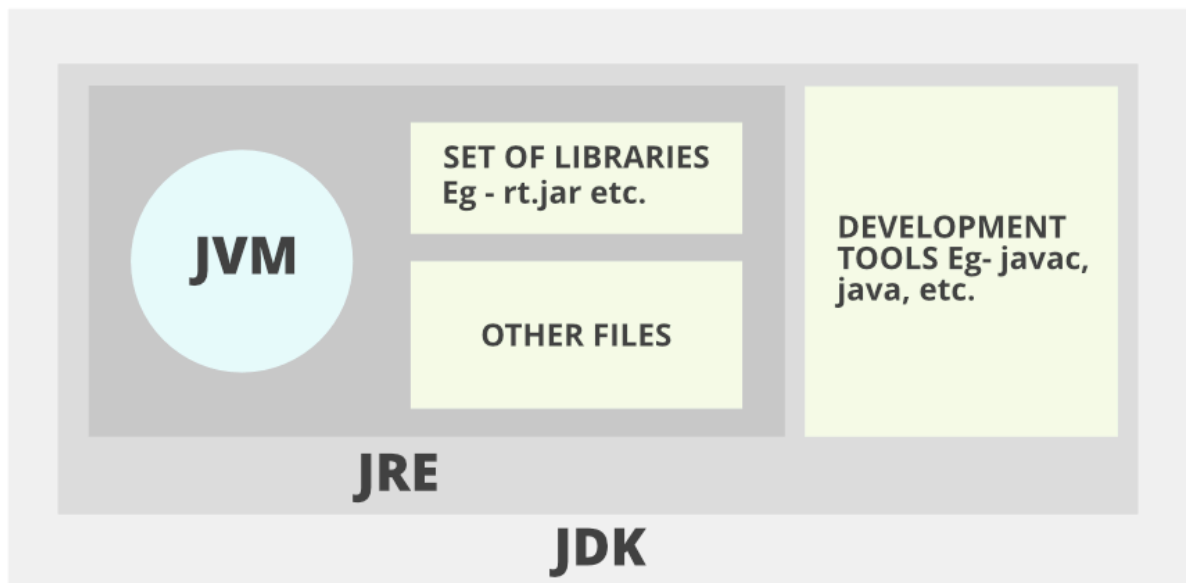
- Java applets (before deprecation) ran in a **restricted environment** using sandboxing.

- Sandbox prevents an applet from:
 - Reading/writing local files.
 - Accessing network resources (except its origin server).
 - Executing system commands.

Differences between JDK, JRE and JVM

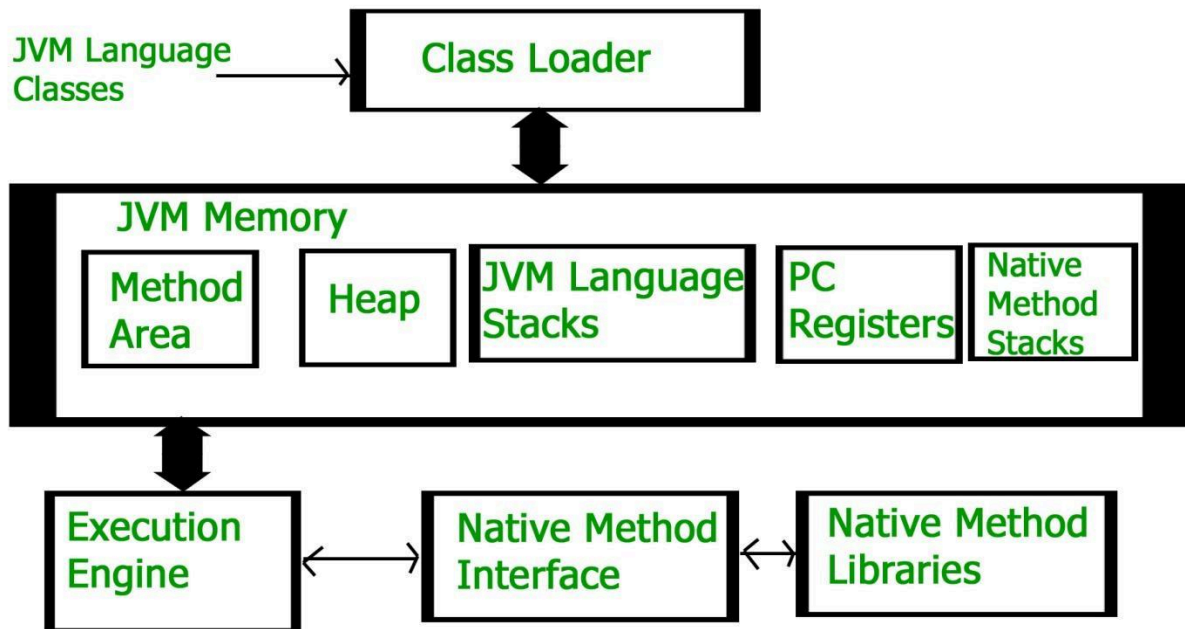
Understanding the **difference between JDK, JRE, and JVM** plays a very important role in understanding how [Java](#) works and how each component contributes to the development and execution of Java applications. The main difference between JDK, JRE, and JVM is:

- [JDK](#): **Java Development Kit** is a software development environment used for developing Java applications and applets.
- [JRE](#): JRE stands for **Java Runtime Environment** and it provides an environment to run only the Java program onto the system.
- [JVM](#): JVM stands for **Java Virtual Machine** and is responsible for executing the Java program.

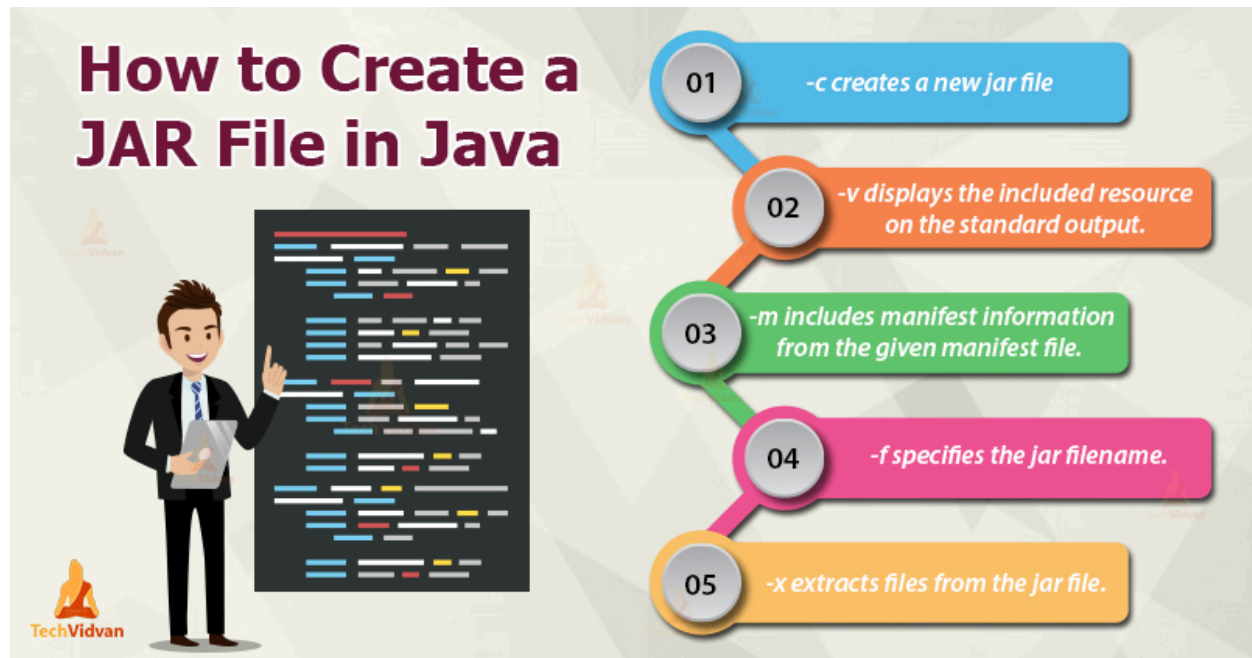


JDK vs JRE vs JVM

Aspect	JDK	JRE	JVM
Purpose	Used to develop Java applications	Used to run Java applications	Responsible for running Java code
Platform Dependency	Platform-dependent	Platform-dependent	Platform-Independent
Includes	It includes development tools like (compiler) + JRE	It includes libraries to run Java application + JVM	It runs the java byte code and make java application to work on any platform.
Use Case	Writing and compiling Java code	Running a Java application on a system	Convert bytecode into native machine code



JAR files in Java



A [JAR \(Java Archive\)](#) is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform.

In simple words, a JAR file is a file that contains a compressed version of .class files, audio files, image files, or directories. We can imagine a .jar file as a zipped file(.zip) that is created by using WinZip software. Even, WinZip software can be used to extract the contents of a .jar . So you can use them for tasks such as lossless data compression, archiving, decompression, and archive unpacking.

Let us see how to create a .jar file and related commands which help us to work with .jar files

1.1 Create a JAR file

In order to create a .jar file, we can use *jar cf command* in the following ways as discussed below:

Syntax:

```
jar cf jarfilename inputfiles
```

Here, cf represents to create the file. For example, assuming our package pack is available in C:\directory, to convert it into a jar file into the pack.jar, we can give the command as:

```
C:\> jar cf pack.jar pack
```

1. 2 View a JAR file

Now, the pack.jar file is created. In order to view a JAR file '.jar' files, we can use the command as:

Syntax:

```
jar tf jarfilename
```

Here, tf represents the table view of file contents. For example, to view the contents of our pack.jar file, we can give the command:

```
C:/> jar tf pack.jar
```

Now, the contents of pack.jar are displayed as follows:

```
META-INF/  
META-INF/MANIFEST.MF  
pack/  
pack/class1.class  
pack/class2.class  
..  
..
```

Here class1, class2, etc are the classes in the package pack. The first two entries represent that there is a manifest file created and added to pack.jar. The third entry represents the sub-directory with the name pack and the last two represent the files name in the directory pack.

Note: When we create .jar files, it automatically receives the default manifest file. There can be only one manifest file in an archive, and it always has the pathname.

```
META-INF/MANIFEST.MF
```

This manifest file is useful to specify the information about other files which are packaged.

1.3 Extracting a JAR file

In order to extract the files from a .jar file, we can use the commands below listed:

```
jar xf jarfilename
```

Here, xf represents extract files from the jar files. For example, to extract the contents of our pack.jar file, we can write:

```
C:\> jar xf pack.jar
```

This will create the following directories in C:\

META-INF

In this directory, we can see class1.class and class2.class.

pack

1.4 Updating a JAR File

The Jar tool provides a 'u' option that you can use to update the contents of an existing JAR file by modifying its manifest or by adding files. The basic command for adding files has this format as shown below:

Syntax:

```
jar uf jar-file input-file(s)
```

Here 'uf' represents the updated jar file. For example, to update the contents of our pack.jar file, we can write:

```
C:\>jar uf pack.jar
```

1.5 Running a JAR file

In order to run an application packaged as a JAR file (requires the Main-class manifest header), the following command can be used as listed:

Syntax:

```
C:\>java -jar pack.jar
```

Data Types, Type Casting, and Operators

1. Data Types in Java

Java has two main types of data types:

Primitive Data Types (8 types)

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127

short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores large whole numbers
float	4 bytes	Stores fractional numbers (6-7 decimal digits)
double	8 bytes	Stores large fractional numbers (15 decimal digits)
char	2 bytes	Stores a single character (Unicode)
boolean	1 bit	Stores true or false

Non-Primitive Data Types

- String
- Array
- Class
- Interface

2. Type Casting in Java

Type casting allows converting one data type into another.

Implicit Casting (Widening)

Automatically converts smaller types to larger types.

```
java
int num = 100; double d = num; // int to double (no data loss)
System.out.println(d); // Output: 100.0
```

Explicit Casting (Narrowing)

Manually converts larger types to smaller types.

```
java
double num = 99.99; int i = (int) num; // double to int (data
loss possible) System.out.println(i); // Output: 99
```

3. Operators in Java

Java supports different categories of operators:

Arithmetic Operators

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (Remainder)	a % b

// Arithmetic Operators

```
int a = 10, b = 5;
```

```
System.out.println("Arithmetic Operators:");
```

```
System.out.println("a + b = " + (a + b)); // Addition -> 15
```

```
System.out.println("a - b = " + (a - b)); // Subtraction -> 5
```

```
System.out.println("a * b = " + (a * b)); // Multiplication -> 50
```

```
System.out.println("a / b = " + (a / b)); // Division -> 2
```

```
System.out.println("a % b = " + (a % b)); // Modulus -> 0
```

Relational (Comparison) Operators

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b

<=	Less than or equal to	a <= b

```
// Relational Operators
System.out.println("\nRelational Operators:");
System.out.println("a == b: " + (a == b)); // false
System.out.println("a != b: " + (a != b)); // true
System.out.println("a > b: " + (a > b)); // true
System.out.println("a < b: " + (a < b)); // false
System.out.println("a >= b: " + (a >= b)); // true
System.out.println("a <= b: " + (a <= b)); // false
```

Logical Operators

Operator	Description	Example
&&	Logical AND	a > 10 && b < 5
`		`
!	Logical NOT	!(a > 10)

```
// Logical Operators
boolean x = true, y = false;
System.out.println("\nLogical Operators:");
System.out.println("x && y: " + (x && y)); // false
System.out.println("x || y: " + (x || y)); // true
System.out.println("!x: " + (!x)); // false
```

Bitwise Operators

Operator	Description	
&	Bitwise AND	
`		
^	Bitwise XOR	
~	Bitwise Complement	
<<	Left Shift	

>>	Right Shift
----	-------------

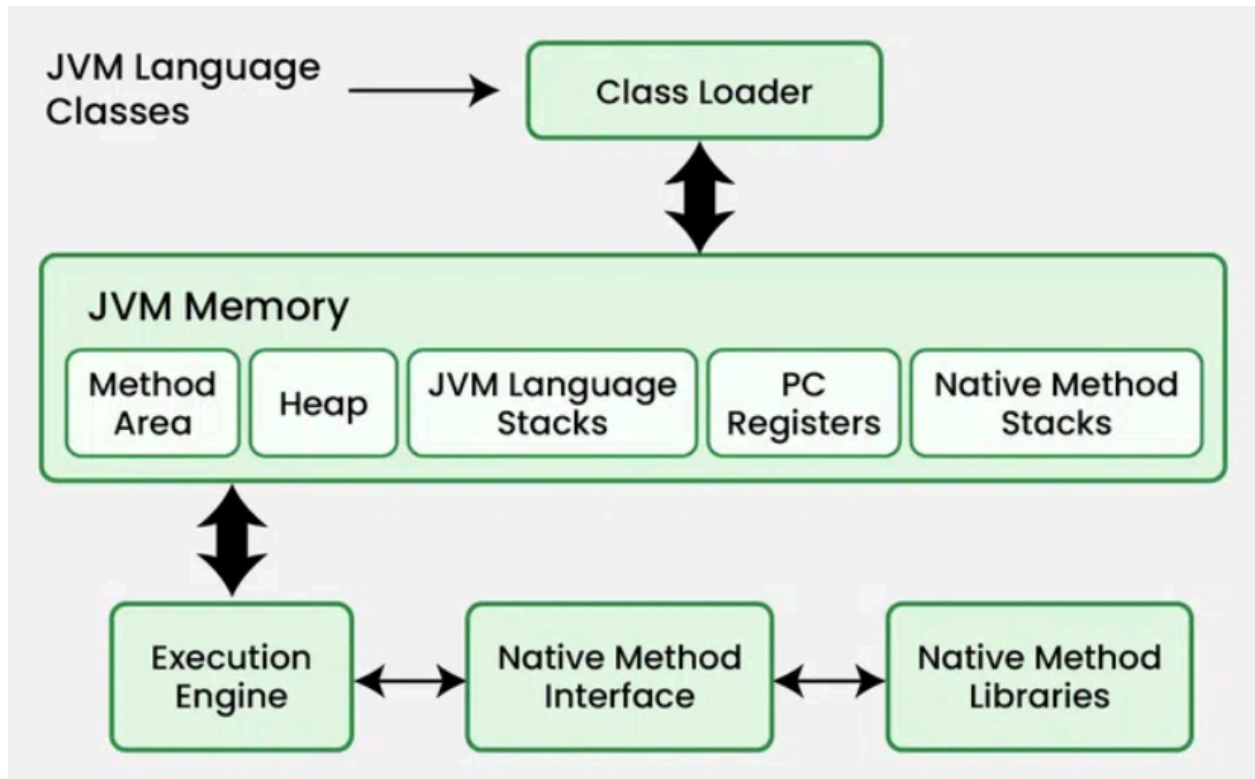
// Bitwise Operators

```
int num1 = 5, num2 = 3;
System.out.println("\nBitwise Operators:");
System.out.println("num1 & num2: " + (num1 & num2)); // 1
System.out.println("num1 | num2: " + (num1 | num2)); // 7
System.out.println("num1 ^ num2: " + (num1 ^ num2)); // 6
System.out.println("num1: " + (num1)); // -6
System.out.println("num1 << 1: " + (num1 << 1)); // 10
System.out.println("num1 >> 1: " + (num1 >> 1)); // 2
```

Security Promises of the Java Virtual Machine (JVM)

The **Java Virtual Machine (JVM)** is designed to provide a **secure execution environment** for Java applications. It ensures that malicious code cannot compromise system security by enforcing strict runtime checks, memory management, and access control mechanisms.

Key Security Promises of JVM



1. Bytecode Verification

- Before execution, the **Bytecode Verifier** checks that the compiled Java code follows the Java language rules.
- Ensures that:
 - No unauthorized memory access occurs.
 - There are no stack overflows or underflows.
 - Method calls match the expected types.
- Prevents **malformed or malicious bytecode** from executing.

2. Automatic Memory Management (Garbage Collection)

- The JVM handles memory allocation and deallocation **automatically**.
- **Prevents memory leaks** and **buffer overflow attacks** (common in languages like C/C++).

3. ClassLoader Security

- The **Java ClassLoader** ensures that only verified and authorized classes are loaded.
- **Prevents Class Conflicts**: Ensures that an untrusted class cannot replace core Java classes.
- **Restricts Untrusted Code**: Separates trusted system classes from untrusted code (e.g., downloaded applets or plugins).

4. Java Security Manager

- **Controls access to system resources** (e.g., file system, network, system properties).
- Prevents unauthorized code from:
 - Reading/writing files.
 - Accessing network resources.
 - Executing system commands.

5. Java Sandbox Model

- Protects **untrusted code** (e.g., Java Applets, Web Start applications) by running it in a **restricted environment**.
- Prevents **unauthorized access to system files, network, and memory**.

6. Cryptographic Security (Java Security APIs)

- JVM provides built-in support for:
 - **Encryption** (`javax.crypto`)
 - **Digital Signatures** (`java.security`)
 - **Secure Random Number Generation** (`java.security.SecureRandom`)
- Ensures **data integrity, authentication, and secure communication**.

7. Access Control & Java Policy Files

- Uses **policy files** to define access permissions for applications.
- Enforced by the **SecurityManager** and **AccessController**.
- Example Policy:

```
grant {  
    permission java.io.FilePermission "/user/docs/*", "read, write";  
    permission java.net.SocketPermission "www.example.com", "connect";  
};
```

8. Protection Against Code Injection & Buffer Overflow

- Unlike C/C++, Java does **not** allow direct memory manipulation (e.g., pointers).
- **Prevents buffer overflow attacks** that can exploit memory vulnerabilities.

9. Secure Class Execution

- Java prevents "**Stack Inspection**" vulnerabilities by ensuring:
 - Secure method execution order.
 - Restricted access to sensitive system functions.

