Java Programming

P by Praveen Kumar

1. Introduction to Java

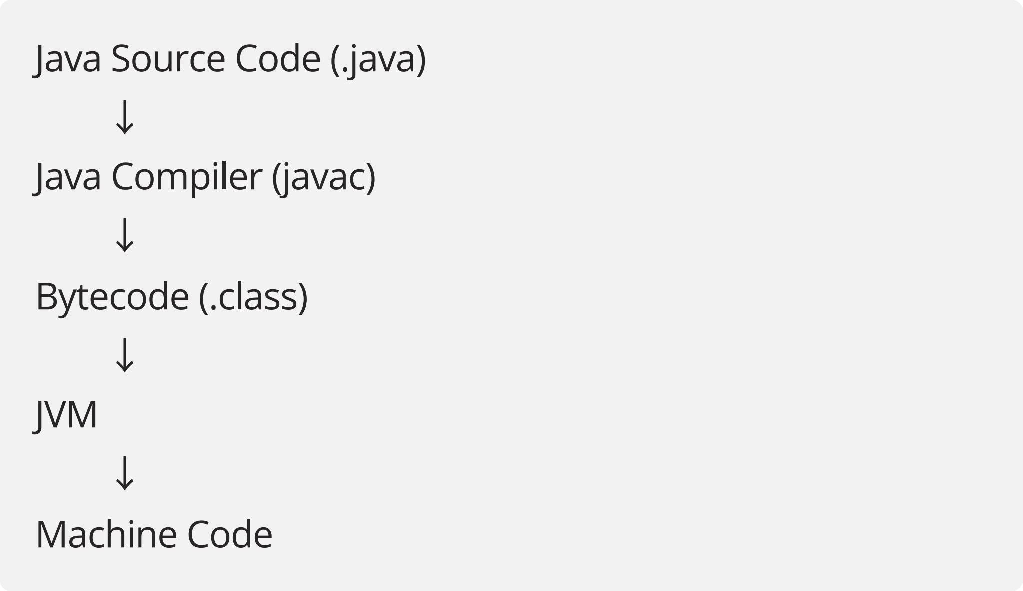
- Java is a **high-level, object-oriented, platform-independent** programming language.
- Developed by **Sun Microsystems** (now Oracle) in 1995.
- Uses **WORA** principle → *Write Once, Run Anywhere*.

Features of Java

- Simple
- Object-Oriented
- Platform Independent
- Secure
- Robust
- Multithreaded
- Portable
- High Performance

2. Java Architecture

Compilation & Execution



Java Source Code (.java)

↓

Java Compiler (javac)

↓

Bytecode (.class)

↓

JVM

↓

Machine Code

JVM Components

- Class Loader
- Memory Areas (Heap, Stack, Method Area)
- Execution Engine
- Garbage Collector

3. Java Environment

- **JDK –** Java Development Kit (compiler + tools)
- **JRE –** Java Runtime Environment
- **JVM –** Java Virtual Machine

What is Java Environment Setup?

Java Environment Setup means **installing and configuring the required software** so that:

- You can **write Java programs**
- **Compile** them
- **Run** them on your computer

To do this, we need **JDK, PATH configuration, and verification**.

JVM (Java Virtual Machine)

- Converts **bytecode into machine code**
- Makes Java **platform independent**
- Handles:
  - Memory management
  - Garbage collection
  - Security

JRE (Java Runtime Environment)

- Required to **run Java programs**
- Includes:
  - JVM
  - Core libraries
- ❌ Cannot compile programs

Steps to Set Up Java Environment

Step 1: Download JDK

- Download the latest **JDK** from Oracle or OpenJDK
- Choose the version according to your OS:
  - Win
  - Linux
  - macOS

Step 2: Install JDK

- Run the installer
- Choose installation directory
  Example:

```
C:\Program Files\Java\jdk
```

Step 3: Set Environment Variables

Environment variables help the OS locate Java tools.

Set JAVA_HOME

- Points to the **JDK installation directory**

Example:

```
JAVA_HOME = C:\Program Files\Java\jdk
```

Set PATH Variable

- Allows running Java commands from **any directory**

Add to PATH:

```
%JAVA_HOME%\bin
```

Steps to Set Variables (Windows):

1. Right-click **This PC → Properties**
2. Click **Advanced system settings**
3. Click **Environment Variables**
4. Under **System Variables**:
   - Add `JAVA_HOME`
   - Edit `Path` and add `%JAVA_HOME%\bin`
5. Click **OK**

Linux / macOS Configuration

Add to `.bashrc` or `.zshrc`:

```
export JAVA_HOME=/usr/lib/jvm/jdk
export PATH=$JAVA_HOME/bin:$PATH

Verify Java Installation

Open Command Prompt / Terminal and type:

java -version
javac -version
```

First Java Program Test

Create a file Hello.java:

```
class Hello {
  public static void main(String[] args) {
    System.out.println("Java Environment Setup Successful");
  }
}

Compile:

javac Hello.java

Run:

java Hello
```

IDE Setup (Optional but Recommended)

Popular Java IDEs:

- Eclipse
- IntelliJ IDEA
- NetBeans

Advantages:

- Auto-completion
- Debugging
- Error detection
- Project management

Java Execution Flow

```
Hello.java → javac → Hello.class → JVM → Output
```

Common Errors & Solutions

Error	Reason	Solution
'java' not recognized	PATH not set	Add JAVA_HOME/bin
Version mismatch	Multiple JDKs	Set correct JAVA_HOME
Compilation error	Syntax issue	Check code

Java Environment Types

- **Development Environment** – JDK installed
- **Runtime Environment** – Only JRE installed
- **Production Environment –** Optimized JRE/JDK

Importance of Java Environment Setup

- Enables Java development
- Ensures platform independence
- Required for running enterprise applications
- Foundation for frameworks like Spring, Hibernate

Integer Data Types (Whole Numbers)

Used to store **whole numbers** (positive, negative, no decimals).

Data Type	Size	Range
byte	1 byte	-128 to 127
short	2 bytes	-32,768 to 32,767
int	4 bytes	-2^{31} to $2^{31}-1$
long	8 bytes	-2^{63} to $2^{63}-1$

2️⃣ Floating-Point Data Types (Decimal Numbers)

Used to store **numbers with decimal points**.

Data Type	Size	Precision
`float`	4 bytes	~6–7 digits
`double`	8 bytes	~15–16 digits

🔹 Example

```
float temperature = 36.5f;
double pi = 3.14159265359;
```

📌 Notes

- `double` is the **default** decimal type
- `float` values must end with **f**

3️⃣ Character Data Type

Used to store a **single character**.

Data Type	Size
`char`	2 bytes

🔹 Example

```
char grade = 'A';
char symbol = '@';
char unicode = '\u0905'; // Unicode for 'अ'
```

📌 Notes

- Uses **Unicode**
- Enclosed in **single quotes**

4️⃣ Boolean Data Type

Used to store **true or false** values.

Data Type	Size
`boolean`	JVM dependent

🔹 Example

```
boolean isJavaFun = true;
boolean isLoggedIn = false;
```

📌 Notes

- Only **true** or **false**
- Cannot use `0` or `1` like C/C++

Default Values of Primitive Data Types

Data Type	Default Value
`byte`	0
`short`	0
`int`	0
`long`	0L
`float`	0.0f
`double`	0.0
`char`	'\u0000'
`boolean`	false

📌 Applies to **instance variables only**, not local variables.

What is a Class Loader?

A **Class Loader** is a **subsystem of the JVM** responsible for:

> **Loading** `.class` **files into memory at runtime**

Java follows:

- **Dynamic class loading**
- Classes are loaded **only when needed**, not all at once

👉 This makes Java **memory-efficient, flexible, and secure**

Why Do We Need Class Loaders?

Without a class loader:

- JVM wouldn’t know **where to find classes**
- No separation between **trusted system classes** and **user-defined classes**
- Security risks (malicious class overriding `java.lang.String`!)

Class Loader provides:

✔️ Dynamic loading
✔️ Security
✔️ Namespace separation
✔️ Platform independence

Class Loader Subsystem – High-Level View

The Class Loader Subsystem performs three main activities:

1. Loading
2. Linking
3. Initialization

Types of Class Loaders (Hierarchy)

Java follows a **parent-first delegation model**.

🔹 1. Bootstrap Class Loader

- **Loads core Java classes**
- Location:

```
<JAVA_HOME>/lib
```

- Loads classes like:
  - `java.lang.*`
  - `java.util.*`
- Written in **native code (C/C++)**
- **No parent**

Example:

```
System.out.println(String.class.getClassLoader()); // null
```

`null` → Loaded by Bootstrap Class Loader

2. Extension Class Loader (Platform Class Loader – Java 9+)

- Loads **extension libraries**
- Location:

```
<JAVA_HOME>/lib/ext
```

- Examples:
  - JDBC drivers
  - XML parsers

```
System.out.println(javax.sql.DataSource.class.getClassLoader());
```

3. Application Class Loader (System Class Loader)

- Loads **user-defined classes**
- Location:
  - CLASSPATH
  - Project `bin` / `target/classes`

```
System.out.println(MyClass.class.getClassLoader());
```

Class Loader Hierarchy

```
Bootstrap ClassLoader
        ↑
Extension / Platform ClassLoader
        ↑
Application (System) ClassLoader
```

Delegation Model (Very Important for Interviews)

Java uses **Parent First Delegation**.

How it works:

1. JVM asks **Application ClassLoader**
2. Application delegates to **Extension ClassLoader**
3. Extension delegates to **Bootstrap ClassLoader**
4. If class not found → comes back down

Why delegation?

✔️ Prevents **duplicate class loading**
✔️ Avoids **security issues**
✔️ Ensures core classes are always trusted

Phase 2: Linking

Linking has **three sub-steps**:

1. Verification

- Bytecode verification
- Stack overflow checks
- Data type validation

👉 Ensures **bytecode safety**

2. Preparation

- Allocates memory for **static variables**
- Assigns **default values**

Example:

```
static int x = 10;
```

During preparation:

```
x = 0
```

3. Resolution

- Converts symbolic references to direct references
- Links methods, fields, interfaces

Phase 3: Initialization

- Executes **static blocks**
- Assigns actual values

```
static int x = 10; // x becomes 10 here
```

7️⃣ Class Loader Example (Simple Demo)

```
public class Test {
    public static void main(String[] args) {
        System.out.println(Test.class.getClassLoader());
        System.out.println(String.class.getClassLoader());
    }
}
```

Output:

```
sun.misc.Launcher$AppClassLoader
null
```