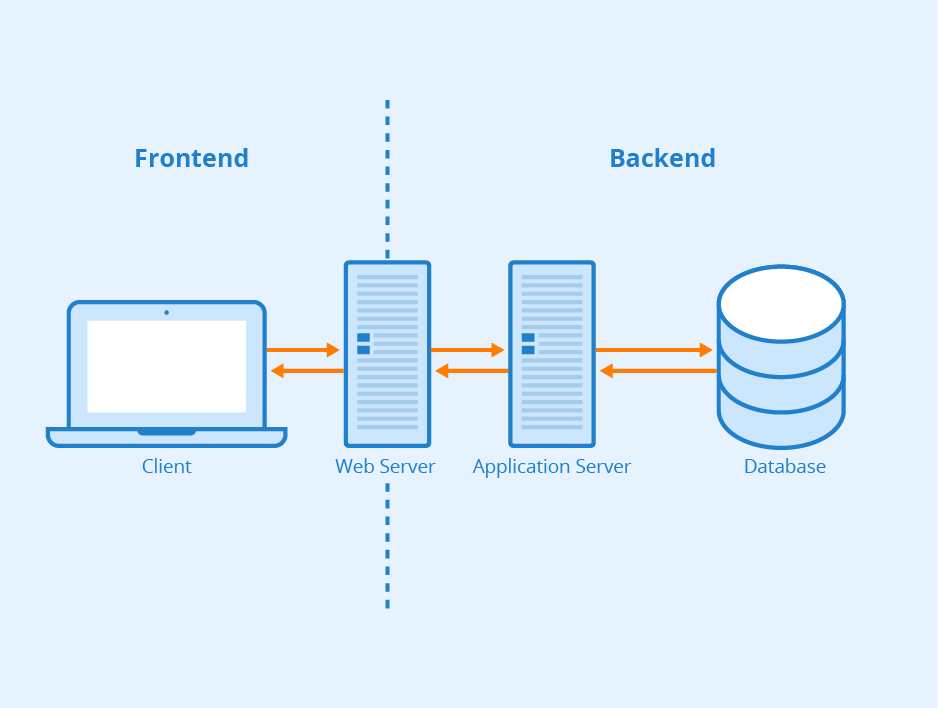
# **What is Java Full Stack**

Java Full-Stack Development refers to building **both front-end (client-side) and back-end (server-side)** applications using Java and related technologies. A **Java Full-Stack Developer** works on everything from UI design to database management, using Java-based frameworks.

A **Java Full-Stack Developer** handles:  
✔ **Front-End** (UI/UX)  
✔ **Back-End** (Server, APIs, Business Logic)  
✔ **Database** (Storing & Retrieving Data)

**Key Components of Java Full-Stack**

| **Layer** | **Technologies** |
| --- | --- |
| **Front-End** | HTML, CSS, JavaScript, React/Angular/Vue.js, Thymeleaf (for Java-based templating) |
| **Back-End** | Java, Spring Boot, Spring MVC, Hibernate, JPA, REST APIs, Microservices |
| **Database** | MySQL, PostgreSQL, MongoDB (NoSQL), Oracle |



**1. Front-End (Client-Side)**

**What it is**:  
The part of an application users **see and interact with** (UI/UX).  
**Key Technologies**:

* **Languages**: HTML (structure), CSS (styling), JavaScript (logic).
* **Frameworks**: React, Angular, Vue.js (for dynamic interfaces).
* **Mobile**: Flutter (Dart), React Native (JavaScript).

**Example**:

* A login form's design, animations, and button clicks are **front-end**.

**Key Responsibilities**:

* Rendering data from the back-end.
* Handling user input (e.g., form validation).
* Communicating with the back-end via **APIs**.

**For Programmers**:

* Uses **HTTP requests** (GET/POST) to fetch/send data to the back-end.
* Modern tools like **Webpack** optimize front-end code.

**2. Back-End (Server-Side)**

**What it is**:  
The "brain" of the application—handles **logic, data processing, and security**.  
**Key Technologies**:

* **Languages**: Java (Spring), Python (Django), JavaScript (Node.js), PHP.
* **Concepts**: APIs, Authentication, Server Deployment.

**Example**:

* When you login, the back-end **verifies your password** and sends your profile data.

**Key Responsibilities**:

* Processing business logic (e.g., payments, calculations).
* Managing **authentication** (e.g., OAuth, JWT tokens).
* Connecting to the **database** to read/write data.

**For Programmers**:

* RESTful APIs (e.g., GET /api/users) or GraphQL.
* Scalability via **load balancers** (e.g., AWS Elastic Beanstalk).

**3. Database**

**What it is**:  
Stores and organizes **application data** (users, products, etc.).  
**Types**:

* **SQL (Relational)**: Structured data (tables).
  + *Examples*: MySQL, PostgreSQL.
  + *Use Case*: Banking systems (strict data integrity).
* **NoSQL (Non-Relational)**: Flexible data (JSON, key-value).
  + *Examples*: MongoDB, Firebase.
  + *Use Case*: Real-time apps (e.g., chat apps).

**Example**:

* Your Facebook posts are stored in a database and fetched when you scroll.

**Key Responsibilities**:

* **CRUD Operations**: Create, Read, Update, Delete data.
* **Optimization**: Indexing for fast queries.

**For Programmers**:

* ORMs (e.g., Hibernate for Java) simplify database interactions.
* Transactions (ACID properties) ensure data consistency.

**How They Work Together**

1. **User** submits a request (e.g., searches for a product).
2. **Front-End** sends the request to the back-end (API call).
3. **Back-End** processes the request (e.g., filters products).
4. **Database** returns matching data to the back-end.
5. **Back-End** sends data to the front-end, which **displays** it.

**Analogy**:

* **Front-End** = Restaurant menu (what you see).
* **Back-End** = Kitchen (prepares your order).
* **Database** = Pantry (stores ingredients).

**Key Differences Summary**

| **Aspect** | **Front-End** | **Back-End** | **Database** |
| --- | --- | --- | --- |
| **Focus** | User Interface | Business Logic | Data Storage |
| **Languages** | HTML/CSS/JavaScript | Java, Python, Node.js | SQL, NoSQL queries |
| **Tools** | React, Figma | Spring Boot, Django | MySQL, MongoDB |
| **Output** | Visual elements | API responses | Query results |

**Why This Matters**

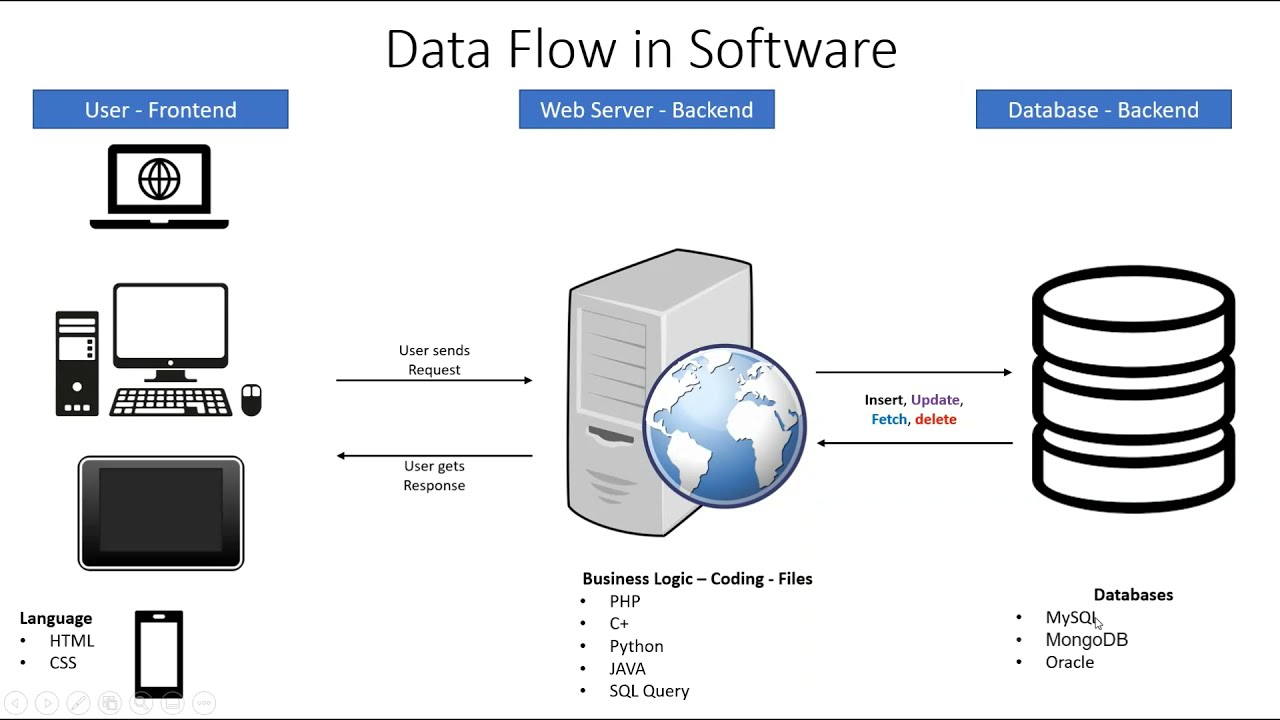
* **Full-Stack Developers** work on **both front-end and back-end**.
* **Specialization**: Front-end devs focus on UX; back-end devs optimize servers.

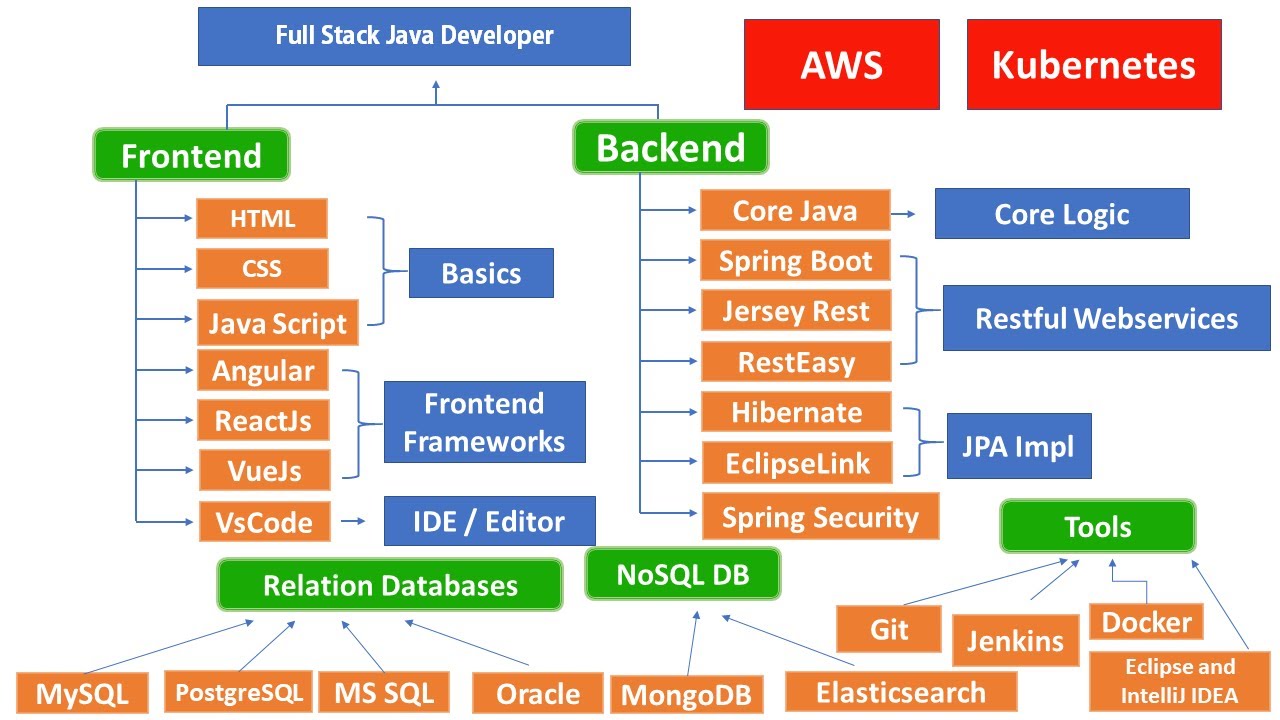
**Major Companies Using Java Full Stack**

1. **Google**: Utilizes Java for backend services and Android app development.
2. **Amazon**: Employs Java for its e-commerce platform and AWS services.
3. **Netflix**: Uses Java for building scalable and reliable backend systems.
4. **LinkedIn**: Relies on Java for its server-side applications.
5. **Uber**: Implements Java for handling large-scale backend operations.
6. **Airbnb**: Uses Java for backend services and data processing.

**Popular Sites Built with Java Full Stack**

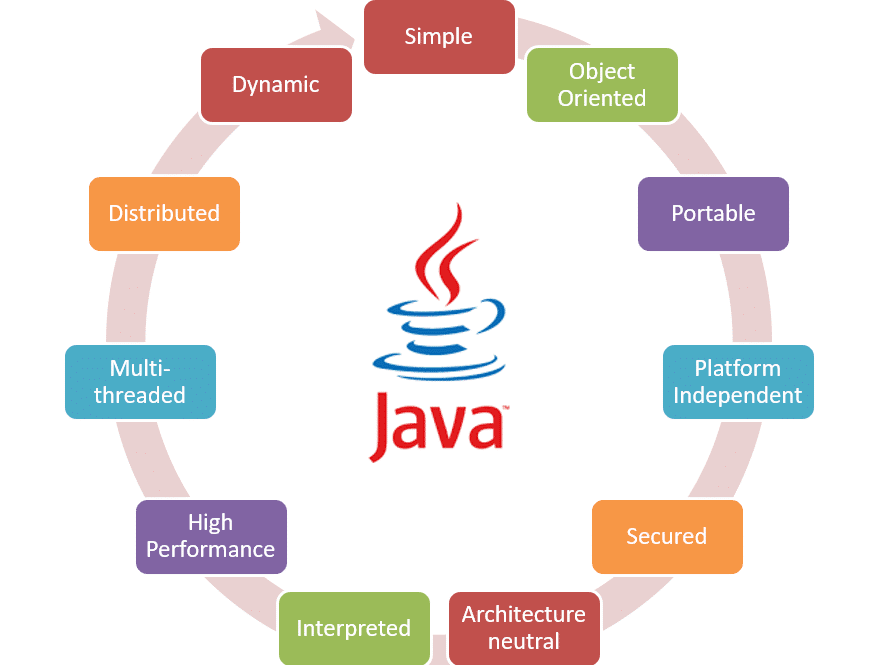
1. **Spotify**: Uses Java for backend services to manage music streaming and user data.
2. **Twitter**: Employs Java for its backend infrastructure to handle high traffic.
3. **eBay**: Relies on Java for its auction and e-commerce platform.
4. **Wikipedia**: Uses Java for search and backend functionalities.
5. **Pinterest**: Implements Java for backend services and data management.





## **Introduction Java**

**Java** is a versatile, high-level, object-oriented programming language designed for **portability, reliability, and scalability**. It powers enterprise applications, Android apps, web services, and big data systems.



1. **"Write Once, Run Anywhere" (WORA)**
   * Java code compiles to **bytecode**, which runs on the **Java Virtual Machine (JVM)**, making it platform-independent.
   * Example: The same .class file works on Windows, Linux, or macOS.
2. **Object-Oriented Programming (OOP)**
   * Built on **classes, objects, inheritance, polymorphism, encapsulation, and abstraction**.
3. **Strong Typing & Memory Management**
   * **Statically typed**: Variables must be declared with a type (e.g., int x = 5;).
   * **Automatic garbage collection**: No manual memory deallocation (unlike C++).
4. **Rich Standard Library**
   * Built-in packages for I/O, networking, data structures (Collections), and multithreading.
5. **Performance**
   * **JIT Compilation**: Bytecode is optimized at runtime for speed.
   * Near C++ in performance for many use cases (but slower than native code).
6. **Multithreading & Concurrency**
   * Built-in support for threads (Thread class, Runnable interface).
   * Used in servers and high-performance applications.
7. **Enterprise & Ecosystem**
   * Dominates **enterprise software** (Spring, Jakarta EE), **Android** (until Kotlin), and **big data** (Hadoop).
   * **Frameworks**: Spring Boot (microservices), Hibernate (database).
8. **Security**
   * **Sandbox environment** (JVM restricts unsafe operations).
   * No pointer arithmetic (reduces vulnerabilities).
   * public static void main: Entry point for execution.
9. **Pros & Cons**
   * ✅ **Pros**: Portable, robust, great for large-scale apps.
   * ❌ **Cons**: Verbosity (boilerplate code), slower startup than C++.

Java applications are of the following different flavor:

1. Desktop based thick client applications. Java applications that uses AWT and Swing as the user interface.
2. Web applications, these are thin client applications. Deployed in an application server and presented to user via a web browser.
3. Enterprise applications, these are distributed applications that follows a Java enterprise specification.
4. Mobile applications, these are Java micro applications that runs on mobile devices.

**Java Version History (1996 - Present)**

Here’s a complete timeline of all major Java releases, including their release years and key features:

**1. Java 5 (1.5) - Major Modernization**

| **Version** | **Release Year** | **Key Features** |
| --- | --- | --- |
| **J2SE 5.0** | 2004 | **Generics**, Autoboxing/Unboxing, Enums, Varargs, Annotations (@Override), for-each loop. |

**2. Java 6 (1.6) - Stability & Performance**

| **Version** | **Release Year** | **Key Features** |
| --- | --- | --- |
| **Java SE 6** | 2006 | Scripting Engine (Nashorn later), JDBC 4.0, JAX-WS (Web Services). |

**3. Java 7 (1.7) - Small but Useful Updates**

| **Version** | **Release Year** | **Key Features** |
| --- | --- | --- |
| **Java SE 7** | 2011 | **try-with-resources**, Strings in switch, Diamond Operator (<>), NIO 2.0. |

**4. Java 8 (1.8) - Revolution (Lambdas & Streams)**

| **Version** | **Release Year** | **Key Features** |
| --- | --- | --- |
| **Java SE 8** | 2014 | **Lambda Expressions**, Stream API, java.time (New Date/Time API), Default Methods in Interfaces. |

**5. Java 9-11 - Modularization & Faster Releases**

| **Version** | **Release Year** | **Key Features** |
| --- | --- | --- |
| **Java SE 9** | 2017 | **Modules (JPMS)**, JShell (REPL), Factory Methods for Collections. |
| **Java SE 10** | 2018 | var (Local-Variable Type Inference). |
| **Java SE 11** | 2018 (LTS) | **First LTS after Oracle’s new release model**, HTTP Client (Standard), var in Lambdas. |

**6. Java 12-17 - Modern Features & LTS**

| **Version** | **Release Year** | **Key Features** |
| --- | --- | --- |
| **Java SE 12** | 2019 | Switch Expressions (Preview), Shenandoah GC (Low-latency garbage collector). |
| **Java SE 13** | 2019 | Text Blocks (Multiline strings, preview). |
| **Java SE 14** | 2020 | instanceof Pattern Matching, Records (Preview). |
| **Java SE 15** | 2020 | Sealed Classes (Preview), Hidden Classes. |
| **Java SE 16** | 2021 | Records, Pattern Matching for instanceof (Final). |
| **Java SE 17** | 2021 (LTS) | **Current LTS**, Sealed Classes (Final), Strong Encapsulation for JDK Internals. |

**7. Latest Versions (Post-Java 17)**

| **Version** | **Release Year** | **Key Features** |
| --- | --- | --- |
| **Java SE 18** | 2022 | UTF-8 by Default, Simple Web Server (jwebserver). |
| **Java SE 19** | 2022 | Virtual Threads (Preview), Structured Concurrency (Preview). |
| **Java SE 20** | 2023 | Scoped Values (Preview), Record Patterns (Preview). |
| **Java SE 21** | 2023 (LTS) | Virtual Threads (Final), Sequenced Collections. |

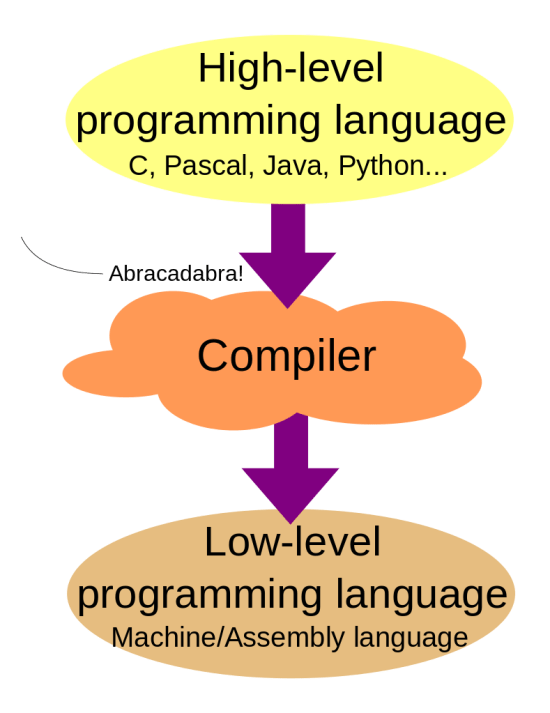
**Key Takeaways**

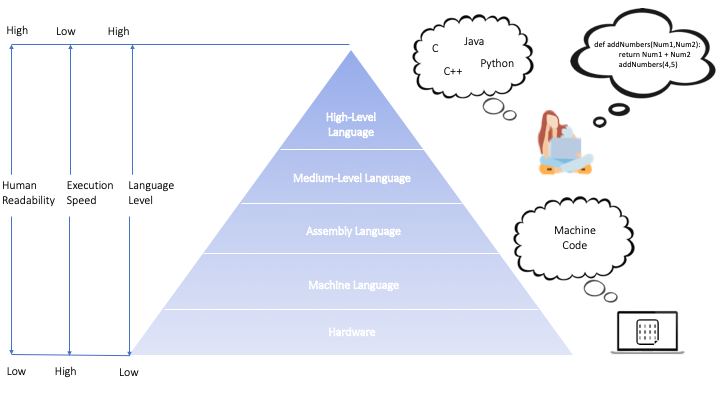
✅ **LTS (Long-Term Support) Versions**: Java 8, 11, 17, 21 (Recommended for production).  
✅ **Major Milestones**:

* **Java 5 (2004)**: Generics, Annotations.
* **Java 8 (2014)**: Lambdas, Streams.
* **Java 11 (2018)**: Modularization, HTTP Client.
* **Java 17 (2021)**: Sealed Classes, Modern Features.

🚀 **Latest Stable Version**: **Java 21 (LTS, 2023)**.

## **Comparison of major languages:**





| **Feature** | **Java** | **C++** | **Python** |
| --- | --- | --- | --- |
| **Type** | *Compiled (to bytecode) + Interpreted (JVM)* | *Compiled* | *Interpreted* |
| **Paradigm** | *Object-Oriented, Imperative, Functional (limited)* | *Object-Oriented, Procedural, Generic, Functional (limited)* | *Object-Oriented, Procedural, Functional, Scripting* |
| **Memory Management** | *Automatic (Garbage Collection)* | *Manual (pointers) + Smart pointers (C++11+)* | *Automatic (Garbage Collection)* |
| **Performance** | *High (JIT optimized)* | *Very High (native compilation)* | *Moderate (interpreted, dynamic typing)* |
| **Syntax Complexity** | *Moderate (verbose)* | *High (complex features like pointers, templates)* | *Low (concise and readable)* |
| **Typing System** | *Static, Strong* | *Static, Strong* | *Dynamic, Strong* |
| **Platform Independence** | *High (Write Once, Run Anywhere - WORA)* | *Low (platform-dependent, recompilation needed)* | *High (interpreted, cross-platform)* |
| **Use Cases** | *Enterprise apps, Android, Web (Spring), Big Data (Hadoop)* | *Game dev, System programming, Embedded systems, High-performance apps* | *Web (Django/Flask), Data Science, AI/ML, Scripting, Automation* |
| **Learning Curve** | *Moderate (OOP concepts, JVM ecosystem)* | *Steep (memory management, pointers, templates)* | *Easy (beginner-friendly syntax)* |
| **Popular Frameworks** | *Spring, Hibernate, Jakarta EE, Android SDK* | *Qt, Boost, STL, Unreal Engine* | *Django, Flask, NumPy, Pandas, TensorFlow, PyTorch* |
| **Community Support** | *Large (enterprise-backed)* | *Large (system dev focus)* | *Very Large (diverse applications)* |
| **Execution Speed** | *Fast (JIT compilation)* | *Very Fast (direct machine code)* | *Slower (interpreted, dynamic typing overhead)* |
| **File Extension** | *.java* | *.cpp, .h* | *.py* |
| **Multithreading** | *Built-in (Thread class)* | *Complex (std::thread, manual management)* | *GIL-limited (threading module), but supports multiprocessing* |
| **Memory Safety** | *Safe (no pointers, bounds checking)* | *Unsafe (manual memory management)* | *Safe (automatic memory management)* |
| **Popularity (2023)** | *#3 (TIOBE)* | *#4 (TIOBE)* | *#1 (TIOBE)* |
| **Companies Using** | *Google, Amazon, Netflix, LinkedIn* | *Google, Microsoft, Gaming (Ubisoft, EA)* | *Google, Facebook, Netflix, Instagram* |
| **Pros** | *Portable, Robust, Multi-threading support* | *High performance, Full control over hardware* | *Easy to learn, Rich libraries, Rapid development* |
| **Cons** | *Slower than C++, Verbosity* | *Complex, Prone to memory leaks* | *Slow execution, Global Interpreter Lock* |

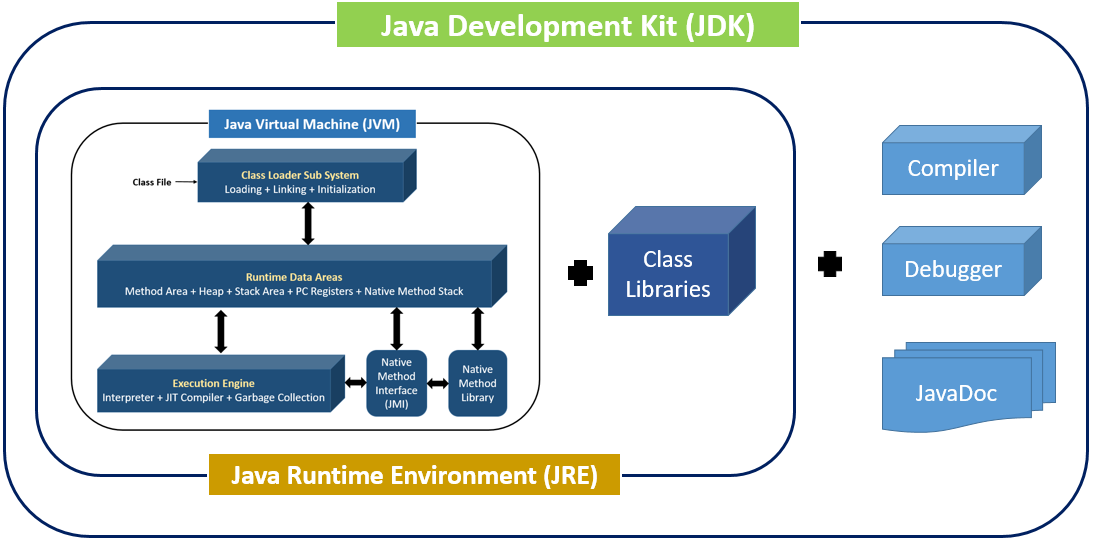
*The TIOBE Programming Community index*

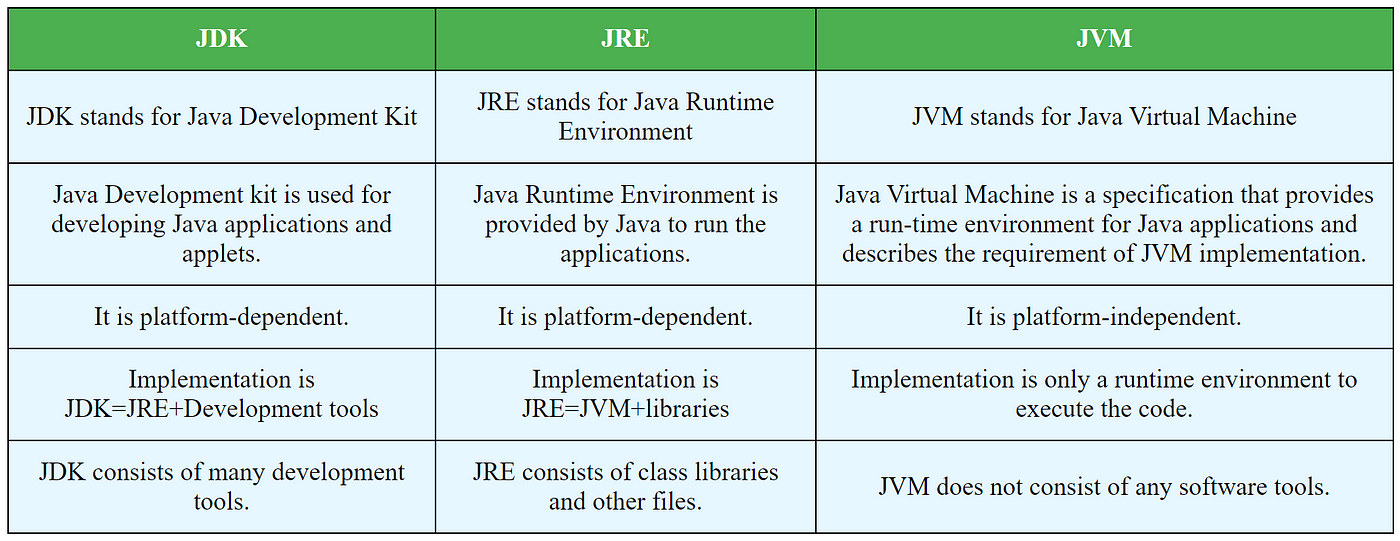
<https://www.tiobe.com/tiobe-index/>

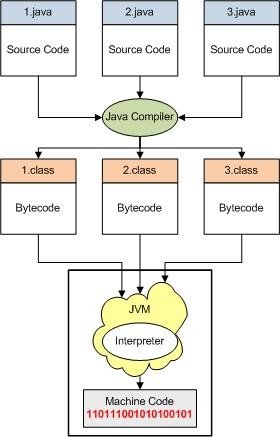
*Popularity of programming languages:*

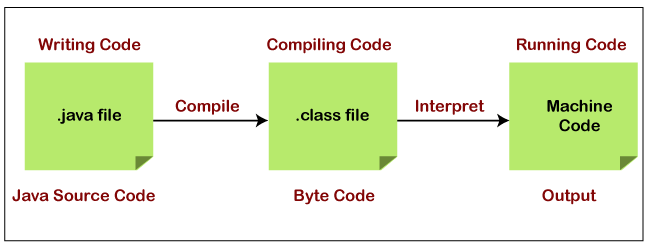
<https://pypl.github.io/PYPL.html>

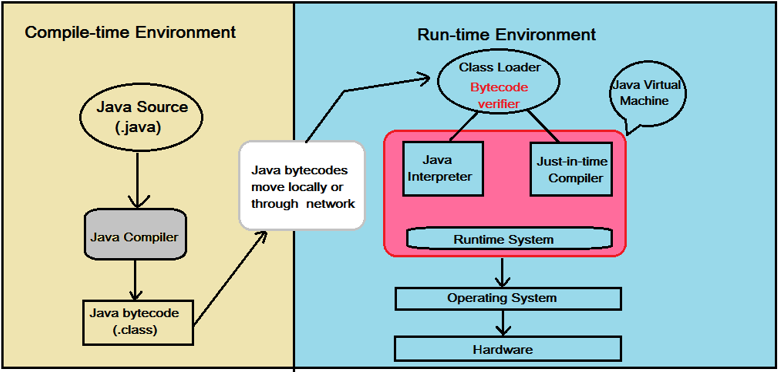
# **What is JVM,JRE, JDK**











**JVM (Java Virtual Machine):**

JVM is the runtime engine that executes Java bytecode, which is generated by the Java compiler. It acts as an abstract computing machine, providing a platform-independent environment to run Java programs. Key features include memory management (garbage collection) and security.  
  
This is automatically used when running any Java application or bytecode. Examples:

* When the JRE executes the bytecode, the JVM kicks in to interpret or JIT-compile it.
* Hosting Java applications on a server where the JVM ensures platform independence.

Every Java application or framework fundamentally relies on JVM during runtime to execute bytecode:  
Tools that Depend on JVM  
**Hadoop**: A big data framework that relies on JVM to process massive datasets.

**Spring Framework**: Used to develop enterprise-level Java applications, the JVM is key to running applications built with Spring.

**Kafka**: A distributed event-streaming platform written in Java, which relies on JVM during runtime.

**JRE (Java Runtime Environment):**

JRE includes everything required to run Java applications. It contains the JVM, libraries, and other files needed for runtime execution. While it enables running Java programs, it does not include tools for development like compilers.  
  
When you're simply running a pre-built Java application but don’t need to write or compile code. For instance:

* Running Minecraft on your computer (if it's a Java edition).
* Executing software like Apache Tomcat or Jenkins, which are Java-based.

These are tools or applications that primarily need JRE to run Java programs without any development features:

**Apache Tomcat**: A web server that runs Java-based web applications, relying on JRE for execution.

**Minecraft Java Edition**: The game runs with JRE to handle its Java-based code.

**Jenkins**: A popular automation server for continuous integration and delivery, it runs on JRE.

**JDK (Java Development Kit):**

JDK is a full-fledged development kit that includes tools for developing Java applications, such as a compiler (javac), debugger, and documentation tools. It also contains JRE, so developers can both write and run Java programs using JDK.

When you're writing and compiling a Java program. For example:

* Developing a web application using Spring Framework.
* Writing Java code for Android apps or desktop applications.

The JDK provides tools like the Java compiler (javac), which translates your source code into bytecode.

Tools that Require JDK  
**Eclipse IDE**: A powerful integrated development environment for writing, debugging, and compiling Java code.

**IntelliJ IDEA**: Another popular IDE for Java developers that uses JDK for code compilation and debugging.

**Summary:**

**JVM**: Executes bytecode.

**JRE**: Runs Java applications (JVM + libraries).

**JDK**: Develops and runs Java applications (JRE + development tools).

# Installation of JAVA and IDE

**Java download:**

<https://www.oracle.com/java/technologies/downloads/#java24>

**Eclipse download:**  
<https://www.eclipse.org/downloads/>

**IntelliJ Download:**

<https://www.jetbrains.com/idea/download>

**PATH (System Environment Variable)**

**What is PATH?**

* The **PATH** is an **operating system-level** environment variable.
* It tells the OS where to look for **executable programs** (like java, javac, git, etc.).

**How PATH Works in Java**

* When you type java or javac in the terminal, the OS searches directories listed in PATH to find these commands.

**Example of Setting PATH (Windows)**

# Add Java's bin directory to PATH

set PATH=%PATH%;C:\Program Files\Java\jdk-17\bin  
  
**Linux/macOS**:

export PATH=$PATH:/usr/lib/jvm/jdk-17/bin

**Real-World Use of PATH**

* Allows running Java commands (java, javac) from **any directory** in the terminal.
* Without PATH, you’d have to specify the full path every time:

**Checking PATH on macOS/Linux**

echo $PATH  
  
**Output Example:**/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/homebrew/bin

**Checking PATH on Windows**

echo %PATH%

**Output Example:**

C:\Windows\system32;C:\Windows;C:\Program Files\Java\jdk-17\bin