Full Stack Java Development - Understanding Java Fundamentals

rohit253.hashnode.dev/full-stack-java-development-chapter-2-fundamentals-of-java-and-its-evolution

Rohit Gawande

Introduction

In this chapter, we will explore the fundamentals of Java, tracing its history from its origins to its current status as a leading programming language. We'll examine its development, how it achieved platform independence, and how it differs from other programming languages like C and C++. This comprehensive guide will include detailed explanations, visuals, and examples to help you understand the evolution and significance of Java in the world of programming.

1. The Evolution of Programming Languages

1.1. Early Programming Languages

Before Java's emergence, programming was dominated by languages like **C** and **C++**. These languages were created in the 1970s and 1980s, respectively, by **Bell Labs** and played a pivotal role in shaping modern programming practices. However, they had limitations, particularly in terms of portability and platform independence.

- **C (1972)**: Developed by Dennis Ritchie at Bell Labs, C was efficient and provided low-level access to memory, making it a powerful language for system programming.
- **C++ (1983)**: Created by Bjarne Stroustrup as an extension of C, C++ added object-oriented programming capabilities but still shared C's limitation regarding portability.

Challenges with Early Languages:

- **Platform Dependency**: Programs compiled in C and C++ were platform-dependent, meaning the compiled code could not be executed on a different platform without recompilation.
- **Non-Portability**: These languages were not portable, as machine-level code generated on one operating system would not run on another without significant modification.

Table 1: Key Features of Early Languages

Language	Year Introduced	Features	Portability	Platform Independence
C	1972	Low-level access, efficient, powerful	Limited	Not Portable
C++	1983	Object-oriented, extensive libraries	Limited	Not Portable

1.2. Introduction of Java

Java was introduced in **1991** by **Sun Microsystems**, founded by **James Gosling** and his colleagues from Stanford University. It was designed to overcome the limitations of earlier languages by being **easy to understand**, **object-oriented**, **portable**, and **platform-independent**.

Java's Key Features:

1. **Object-Oriented**: Java models real-world entities using objects.
2. **Portable**: Java applications can run on any platform that has a **Java Virtual Machine (JVM)**.

3. **Platform Independent**: Following the principle of **"Write Once, Run Anywhere" (WORA)**, Java code can run on any platform without modification.

Portability & Platform Independence with Java:

Java uses a two-step process involving the **Java Compiler (JavaC)** and the **Java Virtual Machine (JVM)**. Instead of compiling directly to machine-level language (MLL) like C and C++, Java compiles to **bytecode**, which is platform-independent. This bytecode can be executed on any platform that has a JVM.

Java's Process for Platform Independence:

1. **Compilation**: Java code is first compiled into bytecode using **JavaC**.
2. **Execution**: The bytecode is interpreted and executed by the JVM on different platforms (Windows, Linux, Mac), without the need for recompilation.

Table 2: Compilation and Execution Scenarios

Compilation Platform	Execution Platform	Result
Windows	**Windows**	Output produced
Windows	**Linux**	No output (platform mismatch)
Linux	**Linux**	Output produced

1.3. Development Timeline

Java 1.0 (Oak):

Java was first introduced in **1991** and officially released in **1995** by Sun Microsystems. The language was originally called **Oak**, but the name was later changed to **Java** after the coffee plant island in Indonesia.

Java gained popularity due to its architecture-neutral approach, which made it suitable for developing internet-based applications. Only the **bytecode** was transported over the network, not the actual code, ensuring security and efficiency. The language was open-source, free to download, and anyone could contribute.

Development Timeline:

Year	Version	Key Features
1991	Introduction	Sun Microsystems, led by James Gosling, introduces Java
1995	Java 1.0	First public release, portable and platform-independent
1996	Oak	Java 1.0 renamed to Java, became architecture-neutral
2011	Acquisition	Oracle acquired Sun Microsystems, continuing the development of Java

Java Naming Origins:

The language was originally named **Oak** by the developers, but it was later renamed to **Java** by the **Green Team**. The name "Java" was inspired by the **coffee plant island in Indonesia**, reflecting the energetic and innovative nature of the language.

Java's Global Impact:

In 1995, Java became widely available as a free, downloadable, open-source language. It revolutionized internet programming by introducing platform independence, making it the go-to language for developing cross-platform applications. The first version, **Java 1.0**, was an architecture-neutral, object-oriented programming language.

Java also introduced an **alpha** and **beta** version before its official release. Its unique portability features, such as **WORA** and JVM-based execution, made it stand out among other programming languages.

By **2011**, Oracle had acquired Sun Microsystems, continuing the legacy of Java as one of the most versatile and widely used programming languages in the world. Java is known for its widespread use in web applications, enterprise systems, and Android app development.

Java marked a significant evolution in programming languages by addressing key challenges faced by earlier languages like C and C++. With its introduction of bytecode and JVM, Java achieved what earlier languages could not—**platform independence and portability**. Its impact on internet programming and cross-platform application development has been profound, and it continues to be a cornerstone of software development.

2. Platform Independence and Portability

Java stands out as a platform-independent and portable language, primarily due to its design and architecture. Let's delve into the concepts of platform dependency, Java's unique approach, and how it achieves portability across different operating systems.

2.1. Understanding Platform Dependency

In programming languages like **C** and **C++**, the compiled code is platform-dependent. This means that if the code is compiled on one platform, such as Windows, it may not run on another platform like Linux without recompilation. This lack of portability arises from how the compiled machine code (MLL) is specific to the processor and operating system.

What is Platform Dependency?

Platform dependency refers to a programming environment where the platform (operating system) used to compile the code must be the same as the one where the program is executed.

For example:

- If you compile a C program on Windows, it produces machine code specific to Windows. To run the program on Linux, you need to recompile it on Linux.
- This is why C and C++ are called **non-portable** languages.

Table 3: Platform Dependency Examples

Compilation Platform	Execution Platform	Result
Windows	Windows	Output produced
Windows	Linux	No output (recompile needed)
Linux	Linux	Output produced

Java, on the other hand, takes a **platform-independent** approach, allowing code to be compiled once and run anywhere, a concept known as **WORA (Write Once, Run Anywhere)**.

2.2. How Java Achieves Platform Independence

Java achieves platform independence through two major components: **bytecode** and the **Java Virtual Machine (JVM)**. Java uses a unique compiler, known as **JavaC**, to compile the high-level code into bytecode, which can be run on any operating system with a JVM.

Java's Compilation and Execution Process:

1. **Java Compiler (JavaC)**: The Java compiler compiles the high-level Java code into **bytecode**, which is platform-independent.
2. **Bytecode**: This intermediate code is not directly understood by the machine but is designed to run on any system using a JVM.
3. **JVM (Java Virtual Machine)**: The JVM is platform-specific, meaning that each operating system has its own version of the JVM. However, the JVM interprets the bytecode and translates it into machine-level language (MLL) for the particular platform.
4. **Platform Independence**: Since the JVM exists for different platforms (Windows, Mac, Linux), the same Java bytecode can be executed on any platform without recompilation.

Table 4: Java's Portability Mechanism

Process	Description
Java (HLL)	Source code written in Java High-Level Language
Bytecode	The compiled Java code is converted into platform-independent bytecode
JVM	The JVM executes bytecode on a platform-specific JVM, converting it to MLL

Java's design allows for portability across different operating systems, while still taking advantage of platform-specific features through the JVM.

2.3. Java's Compilation and Execution Flowchart

To visualize how Java achieves portability, let's break down the flow of how a Java application is compiled into bytecode and executed on various platforms:

Visual 1: Java Portability

Here’s a step-by-step breakdown of how Java code flows through compilation and execution across multiple platforms:

```
Java (HLL) --> JavaC (Compiler) --> Bytecode (Platform-Independent)

Bytecode --> JVM (Windows) --> Machine-Level Language (MLL) --> Windows Processor
--> Output

Bytecode --> JVM (Mac) --> Machine-Level Language (MLL) --> Mac Processor -->
Output

Bytecode --> JVM (Linux) --> Machine-Level Language (MLL) --> Linux Processor -->
Output
```

This flowchart shows how Java compiles the high-level language (HLL) into bytecode. The bytecode is platform-independent, and the JVM for each platform (Windows, Mac, Linux) interprets the bytecode into platform-specific machine code (MLL) that can be executed by the processor.

2.4. Bytecode and JVM

Java uses **bytecode** and a platform-specific **Java Virtual Machine (JVM)** to achieve its "write once, run anywhere" capability. Let’s understand the role of these two components in more detail:

- **Bytecode**: After Java code is compiled, it is converted into bytecode. Bytecode is an intermediate code that can be executed on any platform that has a JVM. Unlike machine-level language, bytecode is not specific to any hardware or operating system. It is platform-independent.
- **JVM (Java Virtual Machine)**: The JVM is responsible for executing bytecode. Each operating system has its own JVM, which converts bytecode into machine-level language (MLL) specific to that platform. JVMs are available for Windows, Mac, Linux, and many other platforms, enabling Java's portability.

Table 5: Bytecode and JVM Interaction

Component	Description
Bytecode	Intermediate, platform-independent code produced by the Java compiler

Component	Description
JVM	Platform-specific virtual machine that converts bytecode into machine-level code

Detailed Flow (Step-by-Step)

1. **Java (HLL)**: You write your Java application in a high-level language (HLL) which is readable by humans but not by machines.
2. **JavaC (Compiler)**: The Java Compiler (JavaC) takes the high-level code and compiles it into bytecode. This bytecode is platform-independent and can run on any machine with a JVM.
3. **Bytecode**: The bytecode is saved into a `.class` file. It is not platform-specific and can be used across different operating systems.
4. **JVM**: The JVM is an interpreter that takes the bytecode and converts it into machine-specific instructions. Each platform (Windows, Mac, Linux) has its own JVM that handles this conversion.
5. **Machine-Level Language (MLL)**: The JVM converts the bytecode into machine-level language, which is specific to the processor and operating system being used.
6. **Processor and Output**: The processor then executes the machine-level instructions, and the program produces the desired output.

Summary of Java's Portability:

Java achieves platform independence by compiling code into platform-independent bytecode, which is executed on platform-specific JVMs. This approach ensures that the same bytecode can run on any platform, making Java portable and an ideal choice for cross-platform applications.

This is why Java is known as a **hybrid language**—it uses both compilation (to bytecode) and interpretation (by the JVM) for execution.

---

3. Bytecode and JVM

3. Bytecode and Java Virtual Machine (JVM)

In this section, we'll delve into two crucial components of the Java ecosystem: **Bytecode** and the **Java Virtual Machine (JVM)**. Understanding how they interact and work together is key to understanding Java's platform independence and execution process.

3.1. Bytecode

Bytecode is an intermediate code produced after compiling Java code. It is not written in a high-level programming language (HLL) nor machine language (MLL), but an intermediary form that can be executed on any platform that has a compatible **JVM**.

Key Features of Bytecode:

- **Platform-independent**: Bytecode can be executed on any machine, regardless of the underlying operating system, as long as the JVM is present.
- **Intermediate Representation**: Bytecode is neither in human-readable form (HLL) nor machine-readable form (MLL). It is an intermediary that allows Java to be platform-neutral.

Table 5: Bytecode and JVM Interaction

Component	Description
Bytecode	Intermediate code produced by the Java compiler, platform-independent.
JVM	Executes bytecode and translates it into machine-specific instructions.

3.2. Java Virtual Machine (JVM)

The **Java Virtual Machine (JVM)** is a platform-specific interpreter that executes the bytecode by converting it into machine language specific to the processor. While the bytecode is platform-independent, the JVM is platform-dependent,

meaning that each platform (Windows, Linux, macOS) has its own version of the JVM.

Key Features of JVM:

- **Platform-dependent**: Each platform requires a separate JVM that translates bytecode into machine language understandable by that platform's hardware.
- **Execution Process**: JVM converts bytecode into **Machine-Level Language (MLL)** using an internal interpreter, which enables the bytecode to be executed on the processor.

Visual 2: JVM Flowchart

```
Java Application
    |
    V
Java Compiler (JavaC)
    |
    V
Bytecode (Platform-Independent)
    |
    V
Java Virtual Machine (JVM)
    |
    V
Machine-Level Language (MLL)
    |
    V
Processor
    |
    V
Output
```

Explanation:

1. **Java Application**: The process starts with a Java program written by the developer. This code is human-readable and must be translated for the machine.
2. **Java Compiler (JavaC)**: The Java compiler takes the human-readable Java code and converts it into **Bytecode**. Bytecode is an intermediate, platform-independent form of the code. This means the same bytecode can be executed on different platforms (Windows, Linux, etc.).

3. **Bytecode (Platform-Independent)**: The compiled bytecode is not specific to any machine or processor. It is designed to be interpreted and executed by the JVM on any platform.

4. **Java Virtual Machine (JVM)**: The JVM interprets the bytecode and converts it into **Machine-Level Language (MLL)**. The JVM is platform-specific, meaning each operating system has its own version of the JVM. However, since the JVM can execute bytecode on any system, it gives Java its platform independence.

5. **Machine-Level Language (MLL)**: The JVM converts the bytecode into machine-specific instructions, also known as machine code or machine-level language, which the processor understands.

6. **Processor**: The machine-level instructions are then executed by the processor, performing the actual computations and tasks described in the original Java program.

7. **Output**: Finally, the processor produces the desired output (e.g., a result displayed on the screen).

This flow explains how Java achieves its "write once, run anywhere" capability, with the JVM playing a critical role in translating bytecode into platform-specific machine code.

3.3. JVM’s Role in Java’s Platform Independence

Even though the **JVM** is platform-dependent, Java achieves platform independence through bytecode, as the JVM handles the conversion of bytecode to platform-specific instructions. This allows Java to be **architecturally neutral** and suitable for **internet-based applications**.

- **JVM Design**: JVMs are designed using the **C language**, and different versions exist for each platform.

- **JDK and JRE**: The **Java Development Kit (JDK)** includes the compiler (JavaC) and JVM, while the **Java Runtime Environment (JRE)** contains just the JVM and libraries needed to run bytecode.

Why Java Takes More Time for Execution Compared to C/C++

Since Java relies on the JVM to interpret the bytecode into machine instructions, the execution process involves an additional layer of translation compared to languages like C or C++, where the code is compiled directly into machine code. This extra step makes Java's execution slower than C/C++.

Flowchart for Java Execution Process

```
Java App --> JavaC --> Bytecode (Platform Independent)
                                    |
                                    V
                                JVM --> MLL --> Processor --> Output
```

This flowchart highlights the process through which Java maintains platform independence while executing the code on different machines.

Conclusion

- **Bytecode** provides platform independence for Java by acting as an intermediate code that can run on any machine with a JVM.
- The **JVM** ensures that the bytecode can be executed on various platforms by converting it into machine language, making Java versatile and widely applicable.
- While Java takes slightly longer to execute than C or C++, its ability to run on any platform makes it one of the most popular programming languages in the world.

4. Compilation vs. Interpretation

Introduction

When it comes to programming, translating high-level code to machine-understandable instructions is essential for execution. This is done using either a **compiler** or an **interpreter**, each following different approaches. In this article, we will explore their differences, working mechanisms, benefits, and examples, all with engaging visuals and tables for better clarity.

Section 4. 1: What is a Compiler?

- **Definition**: A compiler is a program that translates the entire source code of a programming language into machine code in one go. Once compiled, the code can be executed multiple times without the need for recompilation.
- **Process**:

```
Code → Compiler → Compiled Code → Processor (Platform) → Output
```

- **Explanation of Compilation Process**:
  - The compiler takes the entire code as input.
  - It converts the code into machine language (binary or low-level code).
  - The binary code is saved, and once compiled successfully, it can be run on the target platform without re-interpreting the code.
- **Example Languages**: C, C++, Swift, Go.

Section 4.2: What is an Interpreter?

- **Definition**: An interpreter translates code **line by line** into machine code at runtime. If there's an error in the middle of execution, the interpreter stops, but all previously interpreted lines can still work.
- **Process**:

```
Code → Interpreter → Interpreted Code → Processor → Output (Line by Line)
```

- **Explanation of Interpretation Process**:
  - The interpreter reads one line at a time.
  - It translates and executes that line immediately.
  - If an error is encountered at, say, line 99, the interpreter throws an error but still executes lines 1-98.
- **Example Languages**: Python, JavaScript, Ruby.

Section 4.3: Differences Between Compiler and Interpreter

Feature	Compiler	Interpreter
Translation Process	Translates entire code at once.	Translates code line by line.
Error Detection	Detects errors after entire code is compiled.	Detects errors while interpreting line by line.
Execution Speed	Faster execution after compilation.	Slower execution, as interpretation happens at runtime.
Memory Usage	Requires more memory for storing compiled code.	Memory usage is less as no intermediate code is stored.
Recompilation	Code needs to be recompiled after changes.	No need for recompilation; runs directly.
Examples	C, C++, Java (at compile time).	Python, JavaScript, Java (at runtime).

Section 4.4: Hybrid Languages – The Case of Java

- **Java as a Hybrid Language**: Java combines both compilation and interpretation.
    - Java source code (.java file) is compiled into bytecode (.class file) using the **Javac** compiler.
    - Bytecode is then interpreted (or executed) by the **JVM** (Java Virtual Machine) at runtime.
- **Java Process Flow**:

```
  Code (.java) → Compiler (Javac) → Bytecode (.class) → Interpreter (JVM) → Output
```

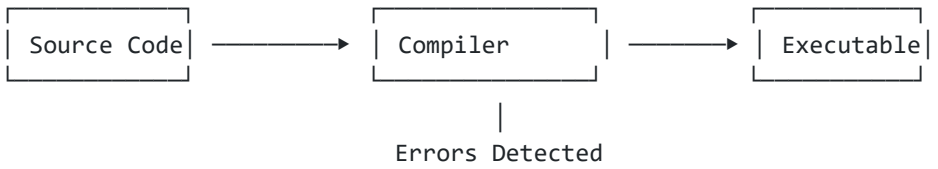
- **Why is Java Hybrid?**
  - Compilation happens at an early stage (producing bytecode).
  - The interpretation occurs during execution (via the JVM).
- **Real-world Use**:
  - Java is one of the most popular languages used in industries.
  - Many companies utilize **Java 8**, which introduced improvements such as lambdas, the Stream API, and more, making it a widely adopted version.

Section 4.5: Industry Example – Java's Dual Nature

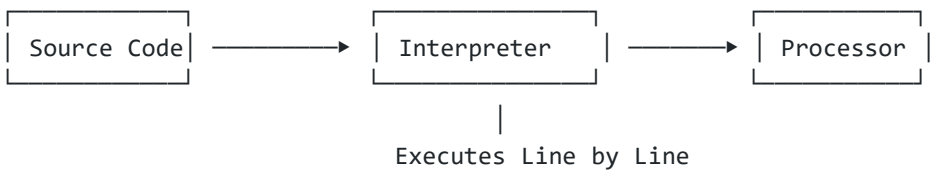
- **Java's Popularity**: Companies prefer Java because it balances performance and flexibility. Compiled bytecode ensures portability across platforms, while the JVM interprets the code efficiently during execution.
- **Java in Production**:
  - Consider a large-scale application where code is written once but can be run on different machines.
  - The compiled bytecode ensures platform independence, and JVM handles execution dynamically.

Section 4.6: In-Depth Process Comparison with Diagrams

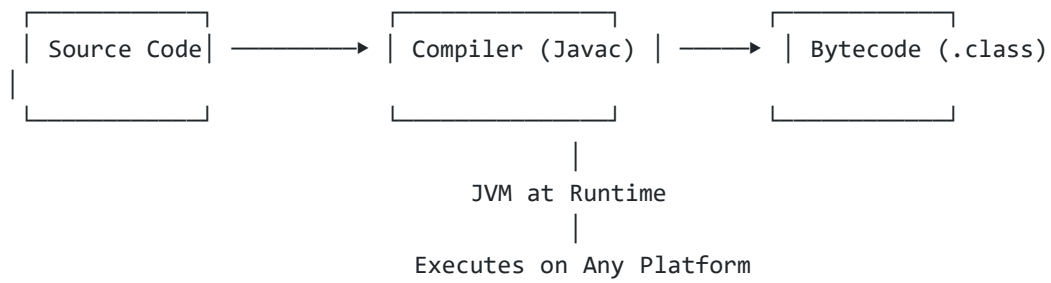
1. **Compiler Flow Diagram**:



2. **Interpreter Flow Diagram**:



3. **Java (Hybrid) Flow Diagram**:



Section 4. 7: Error Handling in Compiler vs Interpreter

- **Compiler**:

  If there's an error in line 99, the entire compilation fails. No executable code is generated until the error is fixed.
- **Interpreter**:

  If there's an error in line 99, the first 98 lines are executed successfully, and the process stops at the error.

Section 4.8: Real-World Applications

1. **Compiler-Dependent Languages**:
   - **C/C++**: Used in system programming, high-performance applications, and embedded systems.
   - **Swift**: Used for iOS development.
2. **Interpreter-Dependent Languages**:
   - **Python**: Used in data science, automation, and machine learning due to its simplicity and dynamism.
   - **JavaScript**: Dominant in web development, primarily used for front-end scripting.
3. **Java as a Hybrid**:

   Java’s versatility makes it suitable for enterprise applications, Android development, and web services.

Section 4.9: Which is Better?

Feature	Compiler Languages (e.g., C/C++)	Interpreter Languages (e.g., Python)	Hybrid Languages (e.g., Java)
Performance	High after compilation.	Slower due to line-by-line execution.	Moderate performance.
Error Handling	All errors must be fixed at once.	Can handle errors line by line.	Balanced approach.
Portability	Platform-specific executable.	High portability (interpreted directly).	Highly portable (bytecode).

Conclusion

Both **compilers** and **interpreters** play crucial roles in programming, and understanding their mechanisms helps developers choose the right tool for the task. Whether it's the speed of compilation or the flexibility of interpretation, each has its place in the software development landscape. **Java**, with its hybrid nature, bridges the gap, offering both compiled performance and interpreted flexibility.

5. Java Versions and Features

5.1. Java Version History

Java has undergone numerous updates since its inception. Here’s a summary of key versions and their features:

Table 6: Java Version Timeline

Version	Year	Key Features
J2SE 1.2	1998	Collection Framework, Swing GUI
J2SE 1.3	2000	Performance improvements
J2SE 1.4	2002	Annotations, Autoboxing, Enhanced for Loop

Version	Year	Key Features
J2SE 5.0	2004	Generics, Metadata annotations, Enumerations
Java SE 6	2006	Performance enhancements, Scripting support
Java SE 7	2011	Try-with-resources, Diamond operator
Java SE 8	2014	Lambda expressions, Stream API, Date/Time API
Java 9	2017	Modular system, JShell
Java 10	2018	Local-variable type inference
Java 11	2018	Long-term support (LTS), New string methods
Java 12	2019	Switch expressions, Multiline strings
Java 13	2019	Text blocks, Updates to Switch expressions
Java 14	2020	Records, Packaging tool
Java 15	2020	Sealed classes
Java 16	2021	More features for records, enhanced pattern matching
Java 17	2021	Long-term support (LTS), Sealed interfaces
Java 18	2022	Virtual threads, Vector API
Java 19	2023	Continued enhancements and updates

5.2. JAR Files

Java Archive (JAR) files are used to bundle multiple .class files into a single file. JAR files facilitate the distribution of Java applications and libraries.

6. Conclusion

Java's journey from its inception to its current state reflects its evolution in meeting the needs of modern programming. With its emphasis on portability, platform independence, and robust features, Java remains a cornerstone of software development.

Feel free to explore more detailed posts on related topics and connect with me for further discussions.

Further Reading and Resources

If you found this post insightful, you might be interested in exploring more detailed topics in my ongoing series:

- **Chapter 1: Fundamentals of Programming**
  An introduction to the different levels of programming languages, including Machine Level Language (MLL), Assembly Level Language (ALL), and High-Level Language (HLL). This chapter explores how these languages work, their differences, and their importance in the evolution of programming.

- **Chapter 3: GitHub**
  A comprehensive guide on version control using GitHub, including setting up repositories, committing changes, and collaborating with others.

- **Chapter 6: Class, Identifiers, Reserved Keywords, Data Types in Java**
  An in-depth look into Java's core building blocks such as classes, identifiers, reserved keywords, and various data types, essential for mastering Java programming.

Additionally, you can explore other series:

- Data Structure and Algorithm**:**
  A detailed series focused on Data Structures and Algorithms using Java, designed to strengthen problem-solving skills.

- Full Stack JavaScript Development
  A complete guide on full-stack development with JavaScript, covering frontend and backend technologies like React, Node.js, and Express.

Connect with Me

Stay updated with my latest posts and projects by following me on social media:

- **LinkedIn**: Connect with me for professional updates and insights.

- **GitHub**: Explore my repositories and contributions to various projects.

- **LeetCode**: Check out my coding practice and challenges.

Your feedback and engagement are invaluable. Feel free to reach out with questions, comments, or suggestions. Happy coding!

Rohit Gawande
Full Stack Java Developer | *Blogger* | *Coding Enthusiast*