Introduction to Programming

What is Programming?

Programming is the process of designing, writing, testing, debugging, and maintaining a set of instructions (called a **program**) that tells a computer how to perform specific tasks.

A programming language is used to communicate with the computer and solve real-world problems efficiently.

"Programming is the process of giving instructions to a computer so that it can perform desired tasks correctly."

"Programming is the process of teaching a computer how to perform tasks by writing instructions in a programming language."

"Programming is the process of creating a sequence of instructions that enables a computer to perform a particular task."

Objectives of Programming

- Solve computational problems.
- Automate repetitive tasks.
- Develop software applications.
- Process and manage data.
- Control hardware devices.
- Improve efficiency and productivity.

Key Components of Programming

Programming typically involves the following steps:

1. Problem Analysis – Understand the problem.
2. Algorithm Design – Develop a logical solution.
3. Flowchart/Pseudocode – Represent the solution.
4. Coding – Write the program in a programming language.
5. Compilation/Interpretation – Convert code into executable form.
6. Testing – Check whether the program works correctly.
7. Debugging – Find and fix errors.
8. Documentation – Explain the program for users and developers.
9. Maintenance – Update and improve the program over time.

Example of Programming

Problem: Display "Hello, World!"

Java Program

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    } }
```

Different Types of Programming Languages

Programming languages can be classified into different generations based on their level of abstraction.

1. Machine Language (1GL)

Machine language consists of binary digits (0s and 1s) that are directly understood by the computer.

Characteristics

- Lowest-level language.
- Executed directly by the CPU.
- Very fast execution.
- Difficult to write and understand.
- Machine dependent.

Advantages

- Fastest execution.
- No translator required.

Disadvantages

- Difficult to learn.
- Error-prone.
- Not portable.

2. Assembly Language (2GL)

Assembly language uses mnemonic codes (such as ADD, MOV, SUB) instead of binary instructions.

Instruction	Purpose	Example
MOV	Copy data	MOV AX, BX

Instruction	Purpose	Example
ADD	Addition	ADD AX, BX
SUB	Subtraction	SUB AX, BX
MUL	Multiplication	MUL BL
DIV	Division	DIV BL
CMP	Compare values	CMP AX, BX
JMP	Unconditional jump	JMP LOOP1
JE	Jump if equal	JE LABEL
JG	Jump if greater	JG LABEL
JL	Jump if less	JL LABEL
LOOP	Repeat loop	LOOP LOOP1
INT	Call interrupt	INT 21H
XCHG	Exchange values	XCHG AX, BX

Example:

Addition:	**Subtraction:**	**Multiplication:**

MOV AL, NUM1
ADD AL, NUM2
MOV RESULT, AL
	MOV AL, 09H
	MOV BL, 04H
	SUB AL, BL
		MOV AL, 06H
		MOV BL, 04H
		MUL BL

Division:	**Swap:**	**Find Largest of Two:**

MOV AX, 20
MOV BL, 5
DIV BL
	MOV AL, 10
	MOV BL, 20
	XCHG AL, BL
		MOV AL, 15
		MOV BL, 20
		
		CMP AL, BL
		JG FIRST
		MOV CL, BL
		JMP END1
		
		FIRST:
		MOV CL, AL
		
		END1:

Characteristics

- Requires an **Assembler** to translate into machine code.
- Easier than machine language.
- Machine dependent.

Advantages

- Easier to understand than binary code.
- Faster than high-level languages.

Disadvantages

- Difficult for large applications.
- Machine dependent.

3. High-Level Language (3GL)

High-level languages use English-like syntax and are easier for humans to understand.

Examples

- Java
- C
- C++
- Python
- C#
- JavaScript
- PHP
- and .NET

Characteristics

- User-friendly.
- Portable.
- Requires a compiler or interpreter.

Advantages

- Easy to write and debug.
- Platform independent (in many cases).
- Supports structured and object-oriented programming.

4. Fourth Generation Language (4GL)

4GLs are designed for database management, report generation, and rapid application development.

Examples SQL, MATLAB, SAS, Oracle Forms, Informix-4GL.

Advantages

- Requires less coding.
- Faster development.
- High productivity.

5. Fifth Generation Language (5GL)

5GLs focus on Artificial Intelligence, machine learning, and logical programming.

Examples

- Prolog
- Mercury
- OPSS

Applications

- Expert Systems
- Robotics
- Artificial Intelligence
- Natural Language Processing

Classification Based on Programming Paradigm

Language Type	Description	Examples
Procedural	Program divided into procedures/functions	C, Pascal
Object-Oriented	Program organized into objects and classes	Java, C++, C#
Functional	Uses mathematical functions	Haskell, Lisp
Scripting	Used for automation and web development	Python, JavaScript, PHP
Logic Programming	Based on logical rules	Prolog

Difference between Compiler & Interpreter

A **Compiler** and an **Interpreter** are language translators that convert source code into machine code, but they work differently.

Feature	Compiler	Interpreter
Translation	Translates the entire program at once	Translates one statement at a time
Execution Speed	Faster after compilation	Slower because translation occurs during execution
Error Reporting	Displays all errors after compilation	Displays one error at a time
Output	Generates an executable/object file	Does not generate a separate executable file
Execution	Program runs after successful compilation	Executes immediately after translation
Memory Usage	More memory required	Less memory required
Debugging	More difficult	Easier because errors are shown line by line
Examples	C, C++, Go	Python, JavaScript, Ruby

Advantages of Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm in which software is organized around **objects** rather than functions.

1. **Code Reusability:** Classes can be reused through inheritance, reducing development time.
2. **Modularity:** Programs are divided into independent classes, making them easier to develop and maintain.
3. **Data Security:** Encapsulation hides data and restricts unauthorized access.
4. **Easy Maintenance:** Changes in one class have minimal impact on other classes.
5. **Flexibility:** Polymorphism allows the same interface to support different implementations.
6. **Scalability:** Large applications can be expanded by adding new classes without affecting existing code.
7. **Easy Debugging:** Independent classes make it easier to locate and fix errors.

8. **Real-World Modeling:** Objects represent real-world entities, making software easier to design and understand.
9. **Improved Productivity:** Reusable code and modular design reduce development time and cost.
10. **Better Software Quality:** OOP promotes organized, maintainable, and reliable code.

Features of Object-Oriented Programming (OOP)

Object-Oriented Programming is based on the concept of objects and classes.

1. Class: A class is a blueprint or template used to create objects.

Example:

```
class Student {
    int rollNo;
    String name;
}
```

2. Object: An object is an instance of a class.

Example:

```
Student s1 = new Student();
```

3. Encapsulation: Encapsulation is the process of binding data and methods together into a single unit (class) and protecting data by restricting direct access.

Benefits

- Data security
- Data hiding
- Better maintainability

4. Abstraction: Abstraction means hiding implementation details and showing only essential features to the user.

Example

- ATM Machine
- Car Driving

In Java, abstraction is achieved using: Abstract Classes and Interfaces

5. Inheritance: Inheritance allows one class to acquire the properties and methods of another class.

Benefits

- Code reuse
- Reduced redundancy
- Easy maintenance

Example

```
class Animal { }

class Dog extends Animal { }
```

6. Polymorphism: Polymorphism means one interface, many forms.

Types

- Compile-Time Polymorphism (Method Overloading)
- Run-Time Polymorphism (Method Overriding)

Benefits

- Flexibility
- Extensibility
- Cleaner code

7. Dynamic Binding: The method to be executed is determined at runtime, enabling runtime polymorphism.

8. Message Passing: Objects communicate with one another by sending messages through method calls.

Example

```
student.display();
```

What is Java?

Java is a high-level, object-oriented, class-based, platform-independent programming language developed by James Gosling and his team at Sun Microsystems in 1995. Today, Java is owned and maintained by Oracle Corporation.

Java was designed with the principle of **"Write Once, Run Anywhere (WORA)"**, meaning that Java programs compiled into bytecode can run on any device that has a Java Virtual Machine (JVM).

Definitions of Java

Different authors and organizations define Java in slightly different ways:

1. **Oracle:**
   Java is a high-level, class-based, object-oriented programming language designed to have as few implementation dependencies as possible.
2. **James Gosling (Creator of Java):**
   Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, and dynamic programming language.
3. **Academic Definition:**
   Java is a general-purpose programming language used to develop desktop, web, enterprise, mobile, cloud, and embedded applications.

Key Features of Java

- **Simple** – Easy to learn and use.
- **Object-Oriented** – Based on objects and classes.
- **Platform Independent** – Runs on any operating system with a JVM.
- **Secure** – Includes features such as bytecode verification and runtime security.
- **Robust** – Strong memory management and exception handling.
- **Multithreaded** – Supports concurrent execution of tasks.
- **Distributed** – Supports network-based applications.
- **Portable** – Consistent behaviour across platforms.
- **High Performance** – Uses the Just-In-Time (JIT) compiler to improve execution speed.

- **Dynamic** – Supports runtime loading of classes and libraries.

Applications of Java

Java is widely used for:

- Desktop applications (e.g., calculators, editors)
- Web applications
- Enterprise software
- Android mobile applications
- Cloud computing
- Banking and financial systems
- Internet of Things (IoT)
- Big data applications
- Scientific and engineering software

Why Java?

Java is one of the world's most popular programming languages because it is simple, reliable, secure, portable, and widely used for developing applications ranging from desktop software to enterprise systems, mobile apps, cloud services, and IoT applications.

Why Was Java Developed?

Before Java, software developed for one operating system often could not run on another without modification. Java was developed to solve this problem by providing platform independence through the Java Virtual Machine (JVM).
Its main objective is "Write Once, Run Anywhere (WORA)."

A Java program is compiled into bytecode, which can be executed on any platform that has a JVM.

Why Learn Java?

Java is worth learning because:

- It is easy to understand and use.
- It supports Object-Oriented Programming (OOP).

- It is platform independent.
- It has strong security features.
- It provides automatic memory management through Garbage Collection.
- It supports multithreading for concurrent programming.
- It has a rich standard library and APIs.
- It is widely used in industry.
- It offers excellent career opportunities.
- It has a large developer community and extensive documentation.

Feature	Benefit
Platform Independent	Runs on Windows, Linux, macOS, and other systems without recompilation
Object-Oriented	Encourages modular, reusable, and maintainable code
Secure	Includes bytecode verification, class loaders, and runtime security features
Robust	Strong exception handling and automatic memory management
Simple	Easier than C++ because it removes features such as pointer arithmetic and multiple inheritance of classes
Multithreaded	Allows multiple tasks to execute simultaneously
Distributed	Suitable for network and internet-based applications
Portable	Produces consistent behaviour across different hardware and operating systems
High Performance	Improved by the Just-In-Time (JIT) compiler
Dynamic	Supports loading classes and libraries at runtime

History behind Java

Java is one of the most popular programming languages in the world. It was developed to create software that could run on different types of devices without requiring modification. The development of Java began in the early 1990s and revolutionised software development through its **"Write Once, Run Anywhere (WORA)"** philosophy.

Timeline of Java History

1991 – The Green Project

- A team of engineers at Sun Microsystems started a research project called the Green Project.
- The project was led by James Gosling, along with Mike Sheridan and Patrick Naughton.
- The goal was to develop a programming language for consumer electronic devices such as televisions, set-top boxes, and home appliances.

1991 – The Oak Programming Language

- James Gosling created a new programming language named Oak.
- An oak tree outside his office inspired the name Oak.
- Oak was designed to be:
    - Simple
    - Platform independent
    - Secure
    - Reliable

However, the name Oak was already registered as a trademark by another company.

1995 – Birth of Java

- The language was renamed Java.
- According to Sun Microsystems, the name was chosen during a brainstorming session and was inspired by Java coffee, reflecting energy and simplicity.
- In 1995, Sun Microsystems officially introduced Java to the public.

Its slogan became:

Write Once, Run Anywhere (WORA)

Why Java Became Popular?

The rapid growth of the Internet in the mid-1990s created a need for portable applications.

Java became popular because it could:

- Run on different operating systems without modification.
- Execute safely inside web browsers (through applets at the time).

- Support networking and distributed computing.
- Reduce development time through code portability.

1996 – Java Development Kit (JDK)

Sun Microsystems released the first Java Development Kit (JDK 1.0), which included:

- Java Compiler (javac)
- Java Virtual Machine (JVM)
- Standard Libraries
- Java Runtime Environment (JRE)

Developers could now write, compile, and run Java programs.

Year	Version	Major Features
1996	JDK 1.0	First official release
1998	JDK 1.2	Swing GUI, Collections Framework
2000	JDK 1.3	Performance improvements
2002	JDK 1.4	Assertions, Regular Expressions
2004	JDK 5.0	Generics, Enhanced for loop, Annotations, Autoboxing
2006	Java SE 6	Performance and web service improvements
2011	Java SE 7	Try-with-resources, Diamond operator
2014	Java SE 8	Lambda Expressions, Stream API, Date-Time API
2017	Java SE 9	Module System (Project Jigsaw)
2018	Java SE 11 (LTS)	Long-Term Support release
2021	Java SE 17 (LTS)	Security, performance, and language enhancements
2023	Java SE 21 (LTS)	Virtual Threads, Pattern Matching, Sequenced Collections

In **2010**, Oracle Corporation acquired Sun Microsystems and became the owner and steward of Java. Oracle continues to release new Java versions and Long-Term Support (LTS) editions.

Difference between C/C++ and Java

C, C++, and Java are all popular programming languages, but they differ in design, execution, memory management, and application areas. The following table highlights the major differences.

Feature	C	C++	Java
Developed By	Dennis Ritchie	Bjarne Stroustrup	James Gosling
First Released	1972	1985	1995
Programming Paradigm	Procedural	Multi-paradigm (Procedural + Object-Oriented)	Object-Oriented (supports some procedural features)
Platform Dependency	Platform dependent	Platform dependent	Platform independent
Compilation	Compiled to machine code	Compiled to machine code	Compiled to bytecode and executed by the JVM
Execution	Directly on the operating system	Directly on the operating system	Through the Java Virtual Machine (JVM)
Memory Management	Manual (malloc(), free())	Manual (new, delete)	Automatic Garbage Collection
Pointers	Fully supported	Fully supported	No direct pointer support
Multiple Inheritance	Not supported	Supported	Not supported for classes (supported through interfaces)
Operator Overloading	Not supported	Supported	Not supported (except + for strings)
Header Files	Required	Required	Uses packages and imports instead of header files
Pre-processor	Supported	Supported	Not supported
Security	Less secure	Less secure	More secure

Exception Handling	Not available	Available	Built-in and extensive
Multithreading	External libraries required	Library support available	Built-in language support
Portability	Low	Moderate	High
Primary Use	System programming	System and application development	Enterprise, web, Android, cloud, IoT

Features of Java

Java is one of the most widely used programming languages because it provides several powerful features that make software development easier, more reliable, and portable.

1. Simple

Java is easy to learn and use because it has a clean syntax. It removes complex features of C++, such as pointer arithmetic, operator overloading (except for strings), and multiple inheritance of classes.

Example:

```
System.out.println("Hello, Java!");
```

2. Object-Oriented

Java is based on the Object-Oriented Programming (OOP) paradigm. Everything is organised into classes and objects.

OOP Principles:

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

Example:

```
class Student {
    String name;
}
```

3. Platform Independent

Java follows the principle:

Write Once, Run Anywhere (WORA)

A Java program is compiled into bytecode, which can run on any operating system with a Java Virtual Machine (JVM).

4. Architecture Neutral

Java bytecode is independent of the underlying processor architecture (such as x86 or ARM), allowing the same program to execute on different hardware platforms with a compatible JVM.

5. Portable

Java programs produce consistent results across different operating systems because data types and libraries are standardised.

6. Secure

Java provides a secure execution environment through:

- Bytecode verification
- Class Loader
- Security Manager (legacy mechanism)
- No direct pointer manipulation
- Runtime security checks

These features help protect systems from many common programming errors and malicious code.

7. Robust

Java is reliable because it includes:

- Strong exception handling
- Automatic garbage collection
- Compile-time and runtime error checking
- Type checking

Example:

```
try {
    int result = 10 / 0;
```

```
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
}
```

8. Multithreaded

Java supports multithreading, allowing multiple tasks to execute concurrently within a single program.

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread Running");
    }
}
```

9. Distributed

Java provides built-in support for developing distributed and network-based applications through APIs such as:

- Networking (java.net)
- Remote Method Invocation (RMI)
- Web services

10. High Performance

Java uses the Just-In-Time (JIT) Compiler, which converts bytecode into native machine code during execution, significantly improving performance.

11. Dynamic

Java can load classes and libraries at runtime, making applications more flexible and extensible.

Example:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

12. Automatic Garbage Collection

Java automatically frees unused memory, reducing memory leaks and simplifying memory management.

Example:

```
Student s = new Student();
s = null;   // Object becomes eligible for garbage collection
```

13. Rich Standard Library

Java provides an extensive collection of pre-built libraries for:

- File handling
- Networking
- Database connectivity (JDBC)
- Collections
- GUI development
- Multithreading
- Security
- XML and JSON processing

Prerequisites before Writing a Java Program

Before you can write, compile, and execute a Java program, certain software and tools must be installed and configured on your computer.

1. Install Java Development Kit (JDK)

The **Java Development Kit (JDK)** is the most important requirement. It contains:

- Java Compiler (javac)
- Java Runtime Environment (JRE)
- Java Virtual Machine (JVM)
- Standard Java Libraries
- Development tools

Recommended Version: Java 21 LTS (or Java 17 LTS)

2. Configure Environment Variables

Set the following environment variables:

JAVA_HOME

Example (Windows):

```
JAVA_HOME = C:\Program Files\Java\jdk-21
```

PATH

Add the JDK's **bin** directory to the PATH.

```
%JAVA_HOME%\bin
```

Verify the installation:

```
java -version
javac -version
```

3. Install an Integrated Development Environment (IDE)

Although Java programs can be written using a simple text editor, an IDE makes development much easier.

Popular Java IDEs include:

- Visual Studio Code (VS Code)
- Eclipse IDE
- IntelliJ IDEA
- Apache NetBeans

4. Install Java Extensions (for VS Code)

If using **Visual Studio Code**, install the following extension:

- **Java Extension Pack**

It includes:

- Language Support for Java
- Debugger for Java
- Test Runner for Java
- Maven for Java
- Project Manager for Java

5. Choose a Text Editor (Optional)

Instead of an IDE, you can write Java code using editors such as:

- Notepad
- Notepad++
- Visual Studio Code
- Sublime Text

However, an IDE is recommended for beginners because it provides syntax highlighting, code completion, debugging, and project management.

6. Create a Working Folder

Create a folder to store Java programs.

Example:

```
JavaPrograms

├── Hello.java
├── Student.java
├── Calculator.java
```

7. Understand the Java Program Structure

Before writing code, you should be familiar with the basic structure of a Java program.

```
public class Hello {

   public static void main(String[] args) {

      System.out.println("Hello Java");

   }

}
```

8. Know Basic Java Syntax

Before programming, understand:

- Keywords
- Identifiers
- Variables
- Data Types
- Operators
- Comments
- Input and Output statements

9. Basic Knowledge of Command Prompt/Terminal

Useful commands:

Navigate to your program folder:

cd JavaPrograms

Compile:

javac Hello.java

Run:

java Hello

10. Verify Java Installation

Run the following commands:

java -version

Example output:

java version "21.0.2"

Compiler version:

javac -version

Example output:

javac 21.0.2

If both commands work correctly, Java has been installed successfully.

Java Virtual Machine (JVM) and Its Significance

What is JVM?

The Java Virtual Machine (JVM) is a software-based virtual machine that provides a runtime environment for executing Java bytecode. It acts as an intermediary between the Java program and the operating system, making Java programs platform independent.

A Java program is first compiled into bytecode (.class files) by the Java compiler (javac). The JVM then loads, verifies, and executes this bytecode.

Definition:

The Java Virtual Machine (JVM) is a virtual runtime environment that converts Java bytecode into machine code and executes it on the host operating system. It enables Java's "Write Once, Run Anywhere (WORA)" capability.

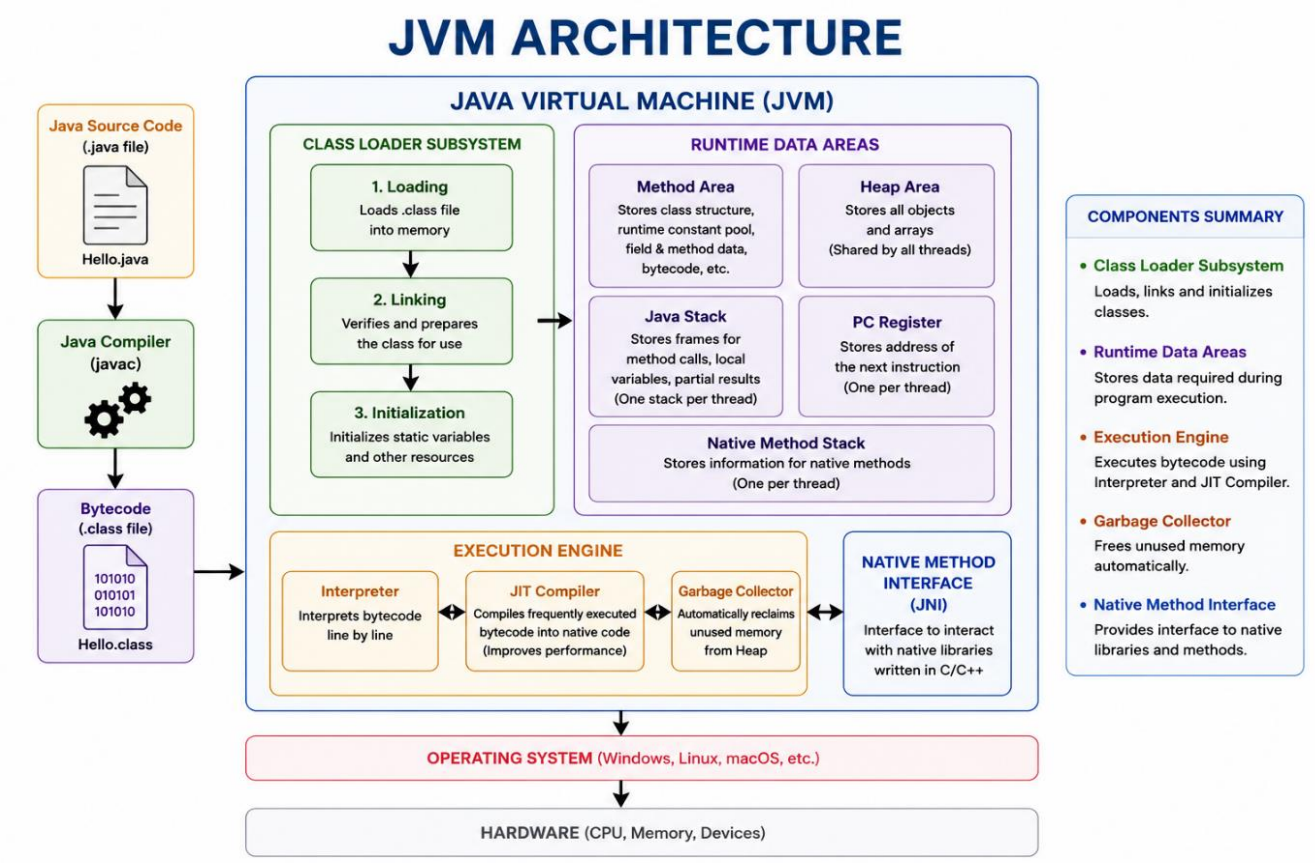
Why Do We Need JVM?

Without the JVM, a Java program would have to be compiled separately for every operating system.

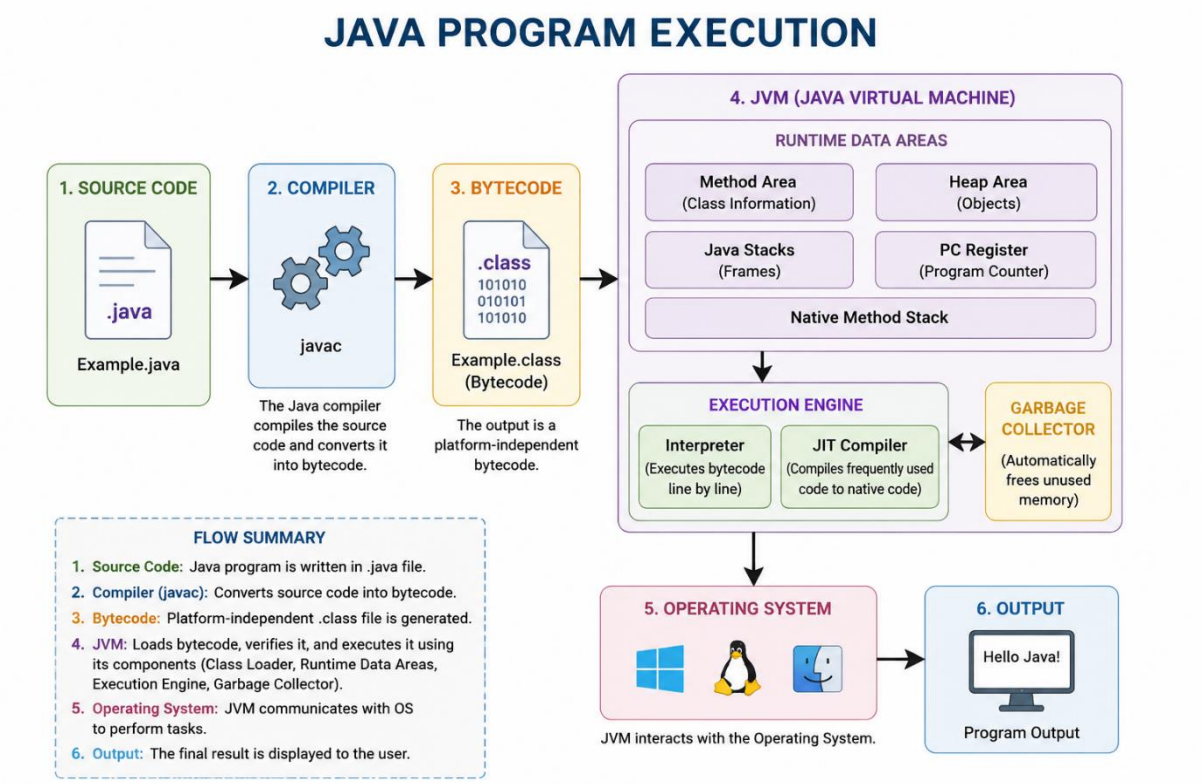
For example:

- C/C++ program compiled on Windows generally cannot run directly on Linux.
- Java program compiled on Windows produces bytecode, which can run on Windows, Linux, macOS, or any other platform with a compatible JVM.

Thus, the JVM eliminates platform dependency.



Java Program Execution Process



Architecture of JVM

The JVM consists of several important components.

1. Class Loader

The Class Loader loads Java class files (.class) into memory.

Its functions include:

- Loading classes
- Linking classes
- Initialising classes

2. Bytecode Verifier

The Bytecode Verifier checks whether the bytecode is valid and secure before execution.

It ensures:

- No illegal memory access
- Valid data types
- Proper stack usage
- No unauthorised code execution

This makes Java more secure.

3. Runtime Data Areas (Memory Areas)

The JVM divides memory into several areas.

a) Method Area

Stores:

- Class information
- Static variables
- Runtime constant pool
- Method bytecode

b) Heap Memory

Stores:

- Objects
- Instance variables

Characteristics:

- Shared by all threads
- Managed by the Garbage Collector

Example:

```
Student s = new Student();
```

The object Student is stored in the Heap.

c) Stack Memory

Each thread has its own stack.

Stores:

- Local variables
- Method calls
- Function parameters

Example:

```
public void add() {
   int x = 10;
}
```

The variable x is stored in the stack.

d) Program Counter (PC) Register

Stores the address of the currently executing JVM instruction for each thread.

e) Native Method Stack

Stores information related to native (non-Java) methods, typically written in languages such as C or C++.

4. Execution Engine

The Execution Engine executes the bytecode.

It contains:

a) Interpreter

- Reads one bytecode instruction at a time.
- Converts it into machine instructions.
- Simple but slower for repeatedly executed code.

b) Just-In-Time (JIT) Compiler

The JIT Compiler improves performance.

Instead of interpreting the same code repeatedly, it:

- Identifies frequently executed ("hot") code.
- Compiles it into native machine code.
- Reuses the compiled code.

This significantly improves execution speed.

Interpreter	JIT Compiler
Executes one instruction at a time	Compiles frequently used code into native code
Slower	Faster after optimisation
Lower initial overhead	Higher initial compilation cost but better long-term performance

5. Garbage Collector (GC)

Java automatically manages memory using Garbage Collection.

It:

- Detects unused objects.
- Frees memory occupied by those objects.
- Helps prevent many memory leaks.

Example:

```
Student s = new Student();
s = null;
```

The object becomes eligible for garbage collection because it is no longer referenced.

Significance (Importance) of JVM

The JVM is the foundation of Java's portability, reliability, and security.

1. Platform Independence

The same Java bytecode can run on different operating systems without recompilation.

Example:

- Windows
- Linux
- macOS

All can execute the same .class file using their respective JVM implementations.

2. Security

The JVM improves security through:

- Bytecode verification
- Class loading mechanism
- Runtime checks

This reduces the risk of executing malformed or malicious code.

3. Automatic Memory Management

The Garbage Collector automatically reclaims memory occupied by unused objects, reducing programmer effort.

4. Improved Performance

The JIT Compiler converts frequently executed bytecode into native machine code, increasing execution speed.

5. Exception Handling

The JVM detects and reports runtime exceptions, helping developers build robust applications.

6. Multithreading Support

The JVM efficiently manages multiple threads, enabling concurrent execution.

7. Reliability

The JVM checks bytecode before execution, reducing crashes caused by invalid instructions.

8. Cross-Platform Compatibility

Developers write the code once, compile it once, and execute it on any platform with a compatible JVM.

Differentiate between JVM, JRE, and JDK

Java Development Kit (JDK)

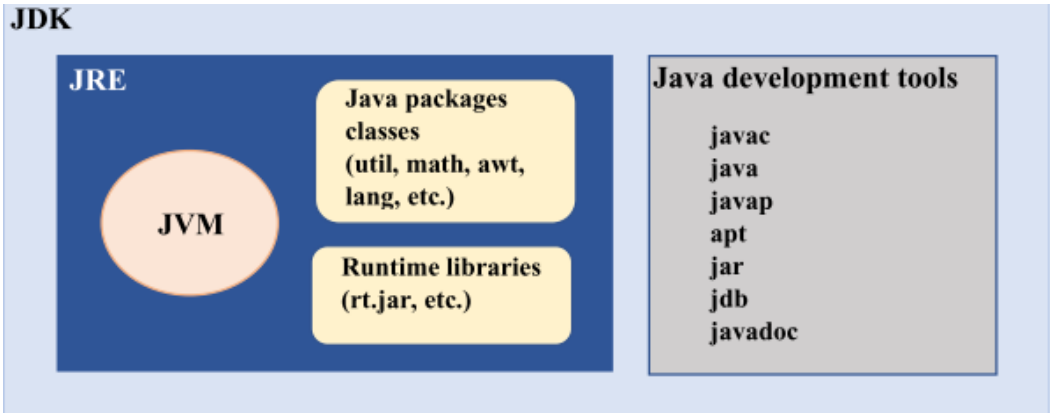
The **Java Development Kit (JDK)** is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.

Java Runtime Environment (JRE)

The Java Runtime Environment provides the minimum requirements for executing a Java application. It consists of the Java Virtual Machine (JVM), java core packages, classes, and supporting files.

Java Virtual Machine (JVM)

The Java Virtual Machine is a specification that provides a runtime environment in which java bytecode can be executed. It means JVM creates a platform to run Java bytecode (.class file) and converting into different languages (native machine language) which the computer hardware can understand. Actually, there is nothing to install as JVM. When the JRE is installed, it will deploy the code to create a JVM for the particular platform. JVMs are available for many hardware and software platforms.



Feature	JVM	JRE	JDK
Purpose	Executes Java bytecode	Provides the runtime environment	Provides tools for Java development
Contains JVM	Yes	Yes	Yes
Contains Java Libraries	No	Yes	Yes
Contains Compiler (javac)	No	No	Yes
Used By	End users and developers	End users	Developers

What are Java Packages?

A **Java package** is a collection of related classes, interfaces, enumerations, and sub-packages that are grouped together to organize Java programs.

Packages help in:

- Organizing code
- Avoiding naming conflicts
- Improving code reusability
- Providing access protection

Types of Packages

A. Built-in (Predefined) Packages

These are provided by Java.

Package	Purpose
java.lang	Basic classes like String, Math, System
java.util	Collections, Scanner, Date, Random
java.io	File handling and input/output
java.net	Networking
java.sql	Database connectivity (JDBC)
java.time	Date and time API
java.awt	GUI development
javax.swing	Swing GUI components

Example:

```
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
    }
}
```

B. User-Defined Packages

Programmers can create their own packages.

Example:

```
package mypackage;

public class Student {
}
```

Advantages of Packages

- Organizes programs
- Prevents class name conflicts
- Makes code reusable
- Improves security
- Simplifies maintenance

What are Java Runtime Libraries?

The **Java Runtime Libraries (Java Class Library)** are a collection of pre-written classes and APIs that provide ready-to-use functionality for Java programs.
Instead of writing everything from scratch, programmers use these libraries.

Common Runtime Libraries

Library	Purpose
java.lang	Fundamental classes
java.util	Utility classes
java.io	File handling
java.nio	Advanced file and buffer operations
java.net	Networking
java.sql	Database programming
java.time	Date and time
java.math	BigInteger, BigDecimal
java.security	Security features
java.xml	XML processing

Examples

String Library

```
String name = "Mukesh";
System.out.println(name.length());
```

Math Library

```
System.out.println(Math.sqrt(25));
```

Output

5.0

Collections Library

```
ArrayList<String> list = new ArrayList<>();
```

Advantages of Runtime Libraries

- Saves development time
- Reliable and well-tested
- Reusable
- Efficient and Platform independent

What are Java Development Tools?

Java Development Tools are programs included in the **JDK** that help developers write, compile, debug, document, package, and monitor Java applications.

Important Java Development Tools

Tool	Description
javac	Compiles Java source code into bytecode
java	Runs Java programs
javadoc	Generates HTML documentation from source code
jar	Creates Java Archive (.jar) files
jdb	Java Debugger
javap	Disassembles .class files
jshell	Interactive Java shell (introduced in Java 9)
jconsole	Monitors Java applications
jps	Lists running Java processes
jstack	Displays thread stack traces
jmap	Displays memory usage
jcmd	Sends diagnostic commands to the JVM

Feature	Java Packages	Java Runtime Libraries	Java Development Tools
Meaning	Organize related classes and interfaces	Pre-written classes and APIs used by programs	Tools used to develop, compile, debug, and execute Java applications
Purpose	Code organization and modularity	Provide reusable functionality	Support the software development process
Included In	JDK/JRE	JRE (and therefore JDK)	JDK only
Examples	java.util, java.io, java.sql	String, Math, ArrayList, Scanner	javac, java, jar, javadoc, jdb

Understanding First Java Program

The first Java program is traditionally the **"Hello World"** program. It helps beginners understand the basic structure of a Java program.

First Java Program

```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello, World!");

    }
}
```

Output:

```
Hello, World!
```

Understanding Each Line

Line 1: public class HelloWorld

Explanation

- **public** → An access modifier that makes the class accessible from anywhere.
- **class** → A keyword used to declare a class.
- **HelloWorld** → The name of the class.
- The class name and file name must be the same.

File Name:

```
HelloWorld.java
```

Line 2: {

Explanation

- Opening curly brace { marks the beginning of the class body.
- Everything inside these braces belongs to the class.

Line 3: public static void main(String[] args)

This is the main() method, which is the entry point of every standalone Java program.

Explanation of Each Keyword

Keyword	Meaning
public	Accessible from anywhere.
static	The method belongs to the class and can be called without creating an object.
void	The method does not return any value.
main	The method name recognised by the JVM as the program's starting point.
String[] args	An array of command-line arguments passed to the program.

Line 4: {

Explanation

- Opening brace of the main() method.
- All executable statements are written inside this block.

Line 5: System.out.println("Hello, World!");

Explanation

This statement prints text on the console.

It can be divided into several parts:

Part	Meaning
System	A predefined class in the java.lang package.
out	A static output stream connected to the console.
println()	Prints the text and moves the cursor to the next line.
"Hello, World!"	The string (text) to be displayed.

Line 6: }

Explanation

- Closing brace of the **main()** method.

Line 7: }

Explanation

- Closing brace of the class.

Why is the main() method static?

Because the JVM calls main() before creating any object. A static method belongs to the class itself, so it can be invoked directly.

Why does main() return void?

Because it does not return any value to the JVM after execution.

Why do we use String[] args?

It allows command-line arguments to be passed to the program.

Example:

```
D:\javap\java HelloWorld John
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello " + args[0]);
    }
}
```

Output:

```
Hello John
```

Java Tokens

Java Tokens are the smallest individual units (building blocks) of a Java program that have a meaningful interpretation to the Java compiler.

A Java token is the smallest meaningful element of a Java program that the compiler recognizes during compilation.

Every Java program is made up of different types of tokens.

Types of Java Tokens: There are five main types of Java tokens:

1. Keywords
2. Identifiers
3. Literals
4. Operators
5. Separators (Delimiters)

1. Keywords

Keywords are reserved words that have predefined meanings in Java. They cannot be used as variable names, class names, or method names.

Java Reserved Keywords

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

_ (underscore)

Java Contextual Keywords

exports	module	open	opens
permits	provides	record	requires
sealed	non-sealed	to	transitive
uses	var	when	with
yield			

Example:

```
public class Student {
    int age;
}
```

Here, public, class, int are **keywords**.

2. Identifiers

Identifiers are names given to:

- Variables
- Methods
- Classes
- Objects
- Packages

Examples

studentName, age, Student, calculateSalary, marks

Example:

```
int age = 20;
```

Here, age is an **identifier**.

Rules for Identifiers

- Must begin with a letter, _ (underscore), or $.
- Cannot start with a digit.
- Cannot be a keyword.
- Case-sensitive.
- May contain letters, digits, _, and $.

Valid Identifiers

Student, student1, _total, $salary

Invalid Identifiers

1student, class, student-name

3. Literals

Literals are fixed values directly written in the program.

Types of Literals

Literal Type	Example
Integer	100, 25
Floating-point	3.14, 12.5
Character	'A', 'X'
String	"Java"
Boolean	true, false
Null	null

Example:

```
int age = 20;
double pi = 3.14;
char grade = 'A';
String name = "Mukesh";
boolean pass = true;
```

Here, 20, 3.14, 'A', "Mukesh", true are literals**.**

4. Operators

Operators perform operations on variables and values.

Types of Operators

Type	Examples
Arithmetic	+, -, *, /, %
Relational	==, !=, >, <, >=, <=
Logical	&&, `
Assignment	= , +=, -=

Type	Examples
Increment/Decrement	++, --
Bitwise	&, `
Conditional (Ternary)	?:

Example:

```
int sum = a + b;
```

Here, =, + are operators**.**

5. Separators (Delimiters)

Separators are symbols used to separate different parts of a Java program.

Common Separators

Separator	Purpose
( )	Parentheses
{ }	Curly braces
[ ]	Square brackets
;	Ends a statement
,	Separates items
.	Accesses members of classes and objects

Example:

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

Separators used: {, }, (,), [,], ., ;

Data Types in Java

What is a Data Type?

A data type specifies the type of data a variable can store. It also determines the size of memory allocated, the range of values, and the operations that can be performed on the data.

Definition:

A data type is a classification that specifies what type of value a variable can store, its memory size, and the range of values it can hold.

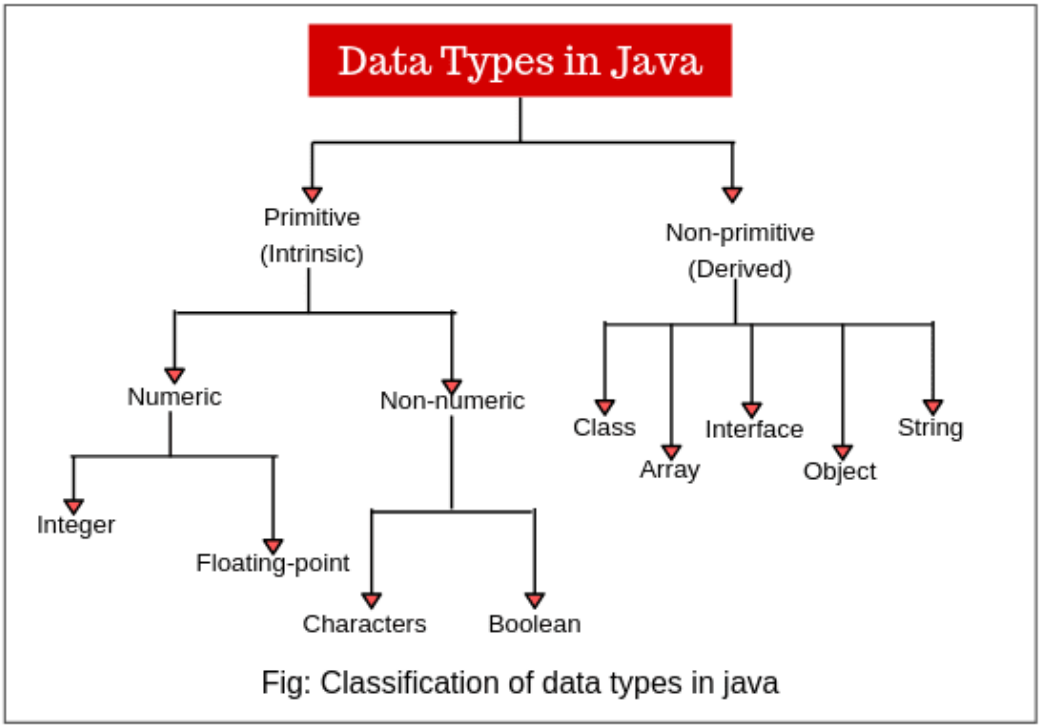


Fig: Classification of data types in java

1. Primitive Data Types

Primitive data types are predefined by Java and store single values.

Java has eight primitive data types.

Data Type	Size	Default Value	Range	Example
byte	1 byte (8 bits)	0	-128 to 127	byte age = 25;
short	2 bytes (16 bits)	0	-32,768 to 32,767	short year = 2025;
int	4 bytes (32 bits)	0	-2^{31} to $2^{31}-1$	int marks = 95;

Data Type	Size	Default Value	Range	Example
long	8 bytes (64 bits)	0L	-2^{63} to $2^{63}-1$	long population = 8000000000L;
float	4 bytes (32 bits)	0.0f	Approx. $\pm 3.4 \times 10^{38}$	float price = 99.5f;
double	8 bytes (64 bits)	0.0d	Approx. $\pm 1.8 \times 10^{308}$	double pi = 3.14159;
char	2 bytes (16 bits)	'\u0000'	Unicode: 0 to 65,535	char grade = 'A';
boolean	JVM-dependent*	false	true or false	boolean pass = true;

Note: Unlike the numeric types, the Java Language Specification does not define a fixed storage size for boolean; it only specifies that it can have the values true or false.

```
public class PrimitiveDemo {
    public static void main(String[] args) {
        byte age = 25;
        short year = 2025;
        int marks = 95;
        long population = 8000000000L;
        float price = 99.5f;
        double pi = 3.1415926535;
        char grade = 'A';
        boolean pass = true;
        System.out.println(age);
        System.out.println(year);
        System.out.println(marks);
        System.out.println(population);
        System.out.println(price);
        System.out.println(pi);
        System.out.println(grade);
        System.out.println(pass);    } }
```

2. Non-Primitive (Reference) Data Types

Non-primitive data types store references (memory addresses) to objects rather than the actual data.

Examples include:

1. String
2. Arrays
3. Classes
4. Objects
5. Interfaces
6. Enums
7. Records

Data Type	Description	Example
String	Stores text	String name = "Mukesh";
Array	Stores multiple values of the same type	int[] marks = {80,90,95};
Class	Blueprint for objects	Student s = new Student();
Object	Instance of a class	Student s = new Student();
Interface	Defines a contract for classes	interface Animal {}
Enum	Represents a fixed set of constants	enum Day {MON, TUE}

Operators in Java

What is an Operator?

An operator is a special symbol that performs a specific operation on one, two, or three operands (values or variables) and produces a result.

An operator is a symbol that tells the Java compiler to perform a specific mathematical, logical, relational, or bitwise operation on one or more operands.

Example

```
int a = 10;
int b = 20;
int sum = a + b;
```

Here,

- a and b are **operands**.
- + is the **operator**.
- = is the **assignment operator**.

Types of Operators in Java

Java operators are classified into the following categories:

Type	Operators
Arithmetic	+, -, *, /, %
Relational	==, !=, >, <, >=, <=
Logical	&&, ||
Assignment	=, +=, -=, *=, /=, %=
Increment/Decrement	++, --
Bitwise	&, |
Shift	<<, >>, >>>
Conditional	?:
Type Comparison	instanceof

1. Arithmetic Operators

Arithmetic operators perform mathematical calculations.

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

2. Relational (Comparison) Operators

Relational operators compare two values and return either true or false.

Operator	Meaning	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

3. Logical Operators

Logical operators combine or negate Boolean expressions.

Operator	Meaning	Example
&&	Logical AND	a > 0 && b > 0
||	Logical OR	a > 0 || b > 0
!	Logical NOT	!(a > b)

Truth Table

A	B	A && B	A || B
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

4. Assignment Operators

Assignment operators assign values to variables.

Operator	Meaning	Example
=	Assign	a = 10
+=	Add and assign	a += 5
-=	Subtract and assign	a -= 5
*=	Multiply and assign	a *= 2

Operator	Meaning	Example
/=	Divide and assign	a /= 2
%=	Modulus and assign	a %= 3

5. Increment and Decrement Operators

Used to increase or decrease a variable by one.

Operator	Meaning
++	Increment
--	Decrement

Types

- **Pre-increment:** ++a
- **Post-increment:** a++
- **Pre-decrement:** --a
- **Post-decrement:** a—

6. Bitwise Operators

Operate on the binary representation of integers.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT

7. Shift Operators

Shift the bits of a number left or right.

Operator	Meaning
<<	Left Shift
>>	Right Shift
>>>	Unsigned Right Shift

8. Conditional (Ternary) Operator

A shorthand form of the if-else statement.

Syntax

condition ? expression1 : expression2;

9. instanceof Operator

Checks whether an object belongs to a particular class or interface.

```
public class OperatorsDemo
{
	public static void main(String[] args)
	{
		int a = 20, b = 10;
		System.out.println("Addition = " + (a + b));
		System.out.println("Subtraction = " + (a - b));
		System.out.println("Multiplication = " + (a * b));
		System.out.println("Division = " + (a / b));
		System.out.println("Remainder = " + (a % b));

		System.out.println(a < b);
		System.out.println(a == b);

		boolean x = true;
		boolean y = false;
		System.out.println(x && y);
		System.out.println(x || y);
		System.out.println(!x);

		int a = 10;
		System.out.println(++a);
		System.out.println(a++);
		System.out.println(a);

		a = 5;
```

```
        b = 3;
        System.out.println(a & b);
        System.out.println(a | b);

        a = 8;
        System.out.println(a << 1);
        System.out.println(a >> 1);

        a = 20;
        b = 30;
        int max = (a > b) ? a : b;
        System.out.println(max);

        String name = "Java";
        System.out.println(instanceof(name));
    }
}
```

Type Casting in Java

The process of converting the value of one data type (int, float, double, etc.) to another data type is known as typecasting. Type casting is also known as type conversion. For example, converting int to double, double to int, short to int, etc.

In Java, there are 13 types of type conversion. However, we will only focus on the major 2 types.

- **Widening Casting** (automatic) - converting a smaller type to a larger type size byte -> short -> char -> int -> long -> float -> double
- **Narrowing Casting** (manual) - converting a larger type to a smaller type size double -> float -> long -> int -> char -> short -> byte

Widening Type Casting

Widening type casting is also known as implicit type casting in which a smaller type is converted into a larger type; it is done by the compiler automatically.

This works because there is no risk of losing information. For example, an int value can safely fit inside a double:

```
public class Main {
  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; //Automatic casting:int to double
    System.out.println(myInt);
    System.out.println(myDouble);
  }
}
```

The compiler plays a role in the type conversion instead of programmers. It changes the type of the variables at the compile time. Also, type conversion occurs from the small data type to large data type only.

Type Conversion Error

The type conversion error occurs when you try to assign a value of a larger data type to a variable of a smaller data type without explicitly converting it.

Java compiler performs implicit type conversion when assigning values between compatible types, but it does not allow the conversion from a larger data type to a smaller one without explicit type casting.

For example:

```
public class Tester
{
      // Main driver method
      public static void main(String[] args)
      {
      // Define int variables
      int num1 = 5004;
      double num2 = 2.5;
      int sum = num1 + num2;
      // show output
```

```
        System.out.println("The sum of " + num1 + " and " +
        num2 + " is " + sum);
        }
    }
```

Compile and run Tester. This will produce the following result –

Output

```
Exception in thread "main" java.lang.Error: Unresolved
compilation problem:
      Type mismatch: cannot convert from double to int
      at com.tutorialspoint.Tester.main(Tester.java:9)
```

Narrowing Type Casting

Narrowing type casting is also known as explicit type casting or explicit type conversion, which is done by the programmer manually. In the narrowing type casting a larger type can be converted into a smaller type.

When a programmer changes the variable type while writing the code. We can use the cast operator to change the type of the variable. For example, double to int or int to double.

Syntax

Below is the syntax for narrowing type casting i.e., to manually type conversion:

```
double doubleNum = (double) num;
```

The above code statement will convert the variable to double type.

```
class Main {
  public static void main(String[] args) {
    // create double type variable
    double num = 10.99;
    System.out.println("The double value: " + num);
    // convert into int type
    int data = (int)num;
    System.out.println("The integer value: " + data);
```

```
    }
}
```

Output

```
The double value: 10.99
The integer value: 10
```

In the above example, we are assigning the double type variable named num to an int type variable named data.

Notice the line,

```
int data = (int)num;
```

Here, the int keyword inside the parenthesis indicates that that the num variable is converted into the int type.

In the case of Narrowing Type Casting, the higher data types (having larger size) are converted into lower data types (having smaller size).

Hence there is the loss of data. This is why this type of conversion does not happen automatically.

Control Statements

Control statements are used in programming to control the flow of execution of a program. They help a program make decisions, repeat tasks, and execute different statements based on conditions.

In Java, control statements can be categorized into the following categories:

- Selection statements (if, switch)
- Iteration statements (while, do-while, for, for-each)
- Jump statements (break, continue, return)

Selection statements

As the name implies, selection statements in Java executes a set of statements based on the value of an expression or value of a variable. A programmer can write several blocks of code and based on the condition or expression, one block can be executed.

Selection statements are also known as conditional statements or branching statements. Selection statements provided by Java are if and switch.

- **if statement:**
  The if statement is used for decision-making in Java. It allows the execution of a block of code only if a specified condition is true.
- **if-else statement:**
  The if-else statement extends the if statement by providing an alternative block of code to be executed if the condition is false.
- **switch statement:**
  The switch statement is used for multi-way branching based on the value of an expression. It provides a concise way to handle multiple possible conditions.

Iteration Statements

While selection statements are used to select a set of statements based on a condition, iteration statements are used to repeat a set of statements again and again based on a condition for finite or infinite number of times.

Iteration statements provided by Java are: for, while, do-while and for-each.

- **for loop:**
  The for loop is used to iterate over a range of values. It includes an initialization, a condition, and an update statement. Java for loop is used to run a block of code for a certain number of times. The syntax of for loop is:

```
for (initialExpression; testExpression;
updateExpression) {

    // body of the loop

}
```

  Here,

  1. The *initialExpression* initializes and/or declares variables and executes only once.
  2. The condition is evaluated in *testExpression*. If the condition is true, the body of the for loop is executed.
  3. The *updateExpression* updates the value of *initialExpression*.

4. The condition is evaluated again. The process continues until the condition is false.

Example 1: Display a Text Five Times

```
class Main {
  public static void main(String[] args) {
    int n = 5;
    // for loop
    for (int i = 1; i <= n; ++i) {
      System.out.println("Java is fun");
    }
  }
}
```

Output

```
Java is fun
Java is fun
Java is fun
Java is fun
Java is fun
```

- **while loop:**

  The while loop repeats a block of code as long as a specified condition is true. It is useful when the number of iterations is unknown.

- **do-while loop:**

  The do-while loop is similar to the while loop, but it ensures that the block of code is executed at least once before checking the condition.

- **Enhanced for loop (for-each):**

  In Java, the **for-each** loop is used to iterate through elements of arrays and collections (like ArrayList). It is also known as the enhanced for loop.

  for-each Loop Syntax: The syntax of the Java for-each loop is:

  ```
  for(dataType item : array) {
    ...
  }
  ```

Here,

- array - an array or a collection
- item - each item of array/collection is assigned to this variable
- dataType - the data type of the array/collection

Example 1: Print Array Elements

```
// print array elements
class Main {
  public static void main(String[] args) {

    // create an array
    int[] numbers = {3, 9, 5, -5};

    // for each loop
    for (int number: numbers) {
      System.out.println(number);
    }
  }
}
```

Output

```
3
9
5
-5
```

Jump Statements

As the name implies, jump statements are used to alter the sequential flow of the program. Jump statements supported by Java are: break, continue and return.

- **break statement:**
  The break statement is used to exit a loop prematurely. It is often used when a certain condition is met, and the loop should terminate immediately.
- **continue statement:**

The continue statement skips the rest of the loop’s code and moves to the next iteration. It is useful when certain conditions should lead to the skipping of specific iterations.

- **return statement:**
  The return statement is used to exit a method and return a value to the calling code.

OBJECT ORIENTED PROGRAMMING USING JAVA

What is JAVA?

- Java is a **programming language** and a **platform**.
- Java is a high level, robust, secured and object-oriented programming language.
- **Platform**: Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform.

Java Example

Let's have a quick look at java programming example. A detailed description of hello Java example is given in next page.

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

Where it is used?

According to Sun Microsystems, 3 billion devices run java. There are many devices where Java is currently used. Some of them are as follows:

- **Desktop Applications** such as acrobat reader, media player, antivirus etc.
- **Web Applications** such as irctc.co.in, javatpoint.com etc.
- **Enterprise Applications** such as banking applications.
- **Mobile**
- **Embedded System**
- **Smart Card**
- **Robotics**
- **Games** etc.

Types of Java Applications

There are mainly 4 types of applications that can be created using java programming:

1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install on every machine such as **media player, antivirus** etc. **AWT** and **Swing** are used in java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

4) Mobile Application

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

Java Platforms / Editions

There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition)

It is a java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java. Math etc. It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection etc.

2) Java EE (Java Enterprise Edition)

It is an enterprise platform which is mainly used to develop web and enterprise applications. It is built on the top of Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA etc.

3) Java ME (Java Micro Edition)

It is a micro platform which is mainly used to develop mobile applications.

4) JavaFx

It is used to develop rich internet applications. It uses light-weight user interface API.

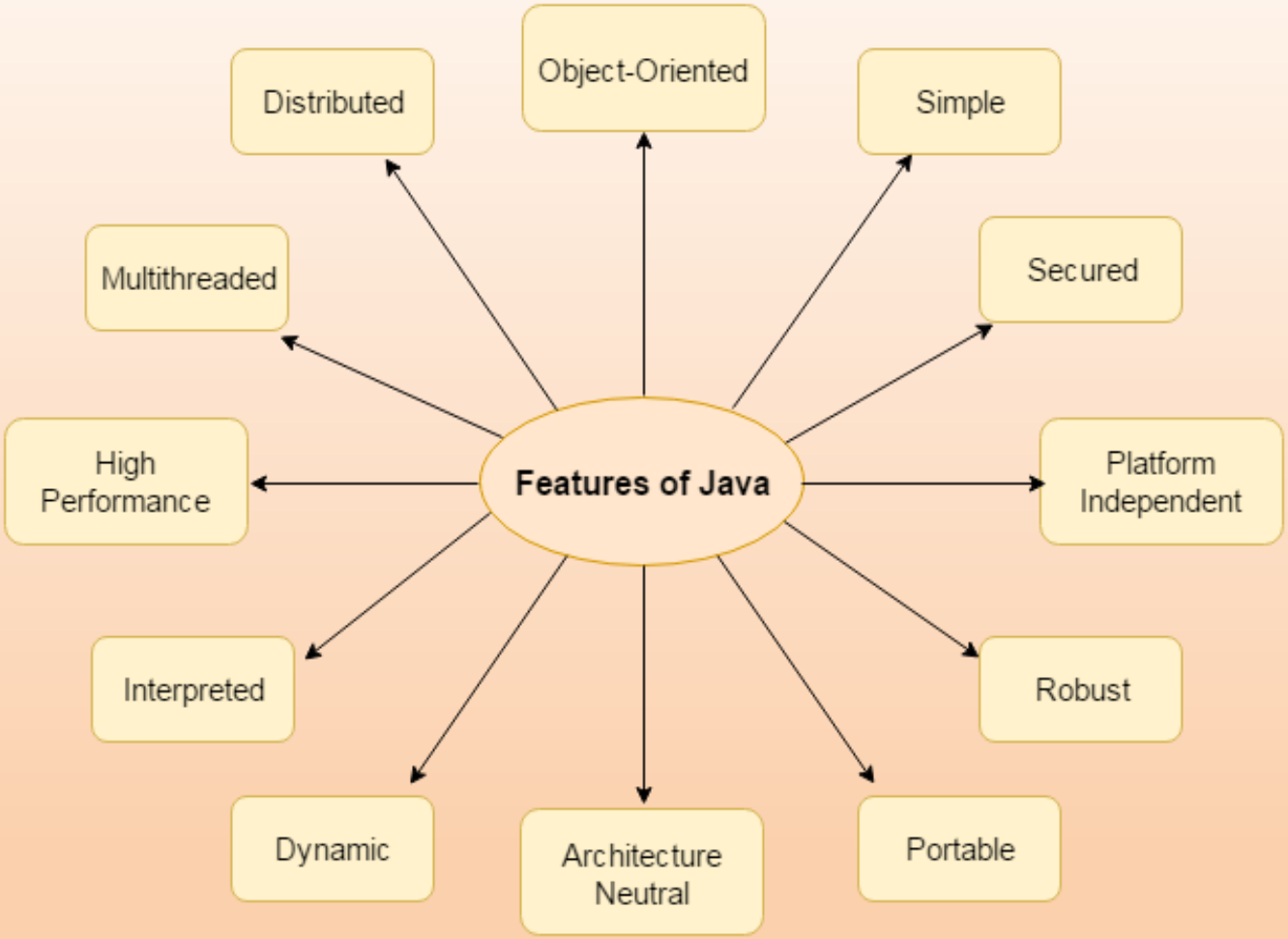
History of Java

- **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
- Originally designed for small, embedded systems in electronic appliances like set-top boxes.
- Firstly, it was called **"Greentalk"** by James Gosling and file extension was ".gt".
- After that, it was called **Oak** and was developed as a part of the Green project.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

Features of Java

- Simple
- Secured
- Platform Independent
- Robust
- Portable
- Architectural Neutral
- Dynamic
- Interpreted
- High Performance
- Multithreaded
- Distributed
- Object Oriented



Features Of java

Simple

- Syntax in based on C++.
- There are no pointers in Java, due to this the application Execution time is improved.
- There is no any concept of Storage Classes, Operator Overloading and Go to Statements and also Java Doesn’t support Multiple Inheritance in the case of classes.
- There is an Automatic Garbage Collector so we do not need to write code manually to Remove unreferenced objects: Which are no further required.

Features Of java

Portable

Java is portable because of the following facts:

- Portable means able to be easily carried or moved, so Java programs are portable means you can write at one platform(Hardware +Operating System) and can carry to and run on the platforms which support JVM, that's why it is called Write Once Run Anywhere(WORA).
- Java is portable due to its JVM.
- It is the combination of both Platform independent + Architecture neutral.

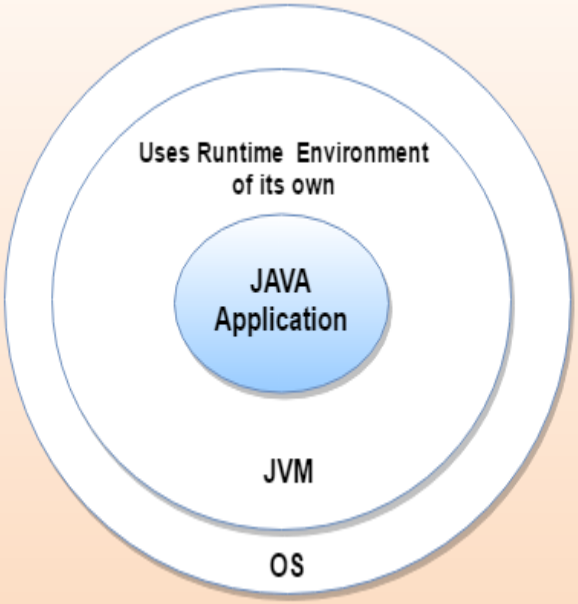
Secure

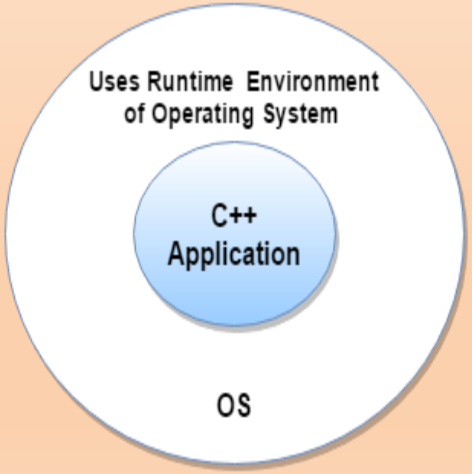
Java is Secure because of the following facts:

- Java is secure language, It enables us to develop virus-free, temper free Systems.

Features Of java

- First Java code is compiled into an intermediate code called bytecode after that bytecode is converted to machine code by JVM.
- Java programs always run in Java JVM environment, hence it is more secure.
- The **JVM** prevents java source code from generating side effects outside of the system.
- **Classloader**: It dynamically loads Java classes which consist on bytecodes into Java Virtual Machine to execute to applications.
- **Bytecode Verifier:** After classloader loads the bytecode into JVM, these bytecodes are first inspected by Bytecode Verifier for Security reasons.

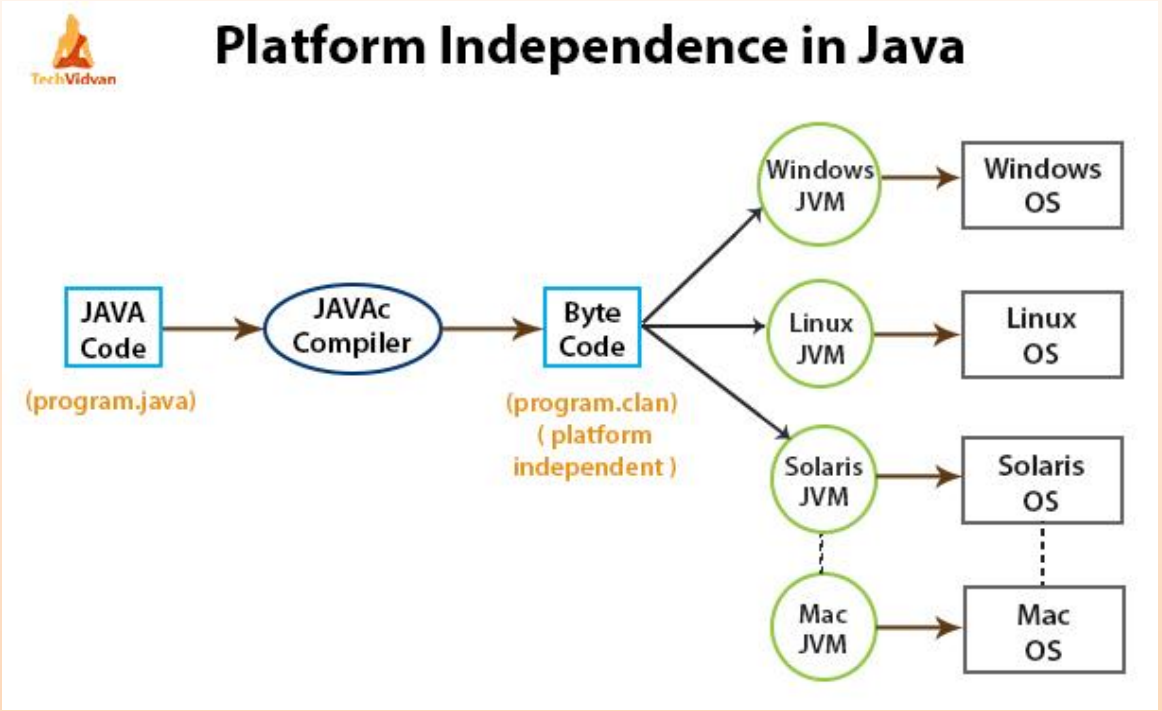




Features Of java

Platform Independent

- Simply a Platform is a hardware or Software platform in which programs runs.
- Platform Independent mean you can execute any Java program on any platforms which support JVM.



- Java source code can be executed on multiple platforms e.g. Linux, Windows, Mac/OS, Sun Solaris, etc. Java code is compiled by the Java compiler and then converted into bytecode. This intermediate bytecode is a platform-independent code because it can be run on a different type of platforms i.e. Write Once and Run Anywhere(WORA).

Features Of java

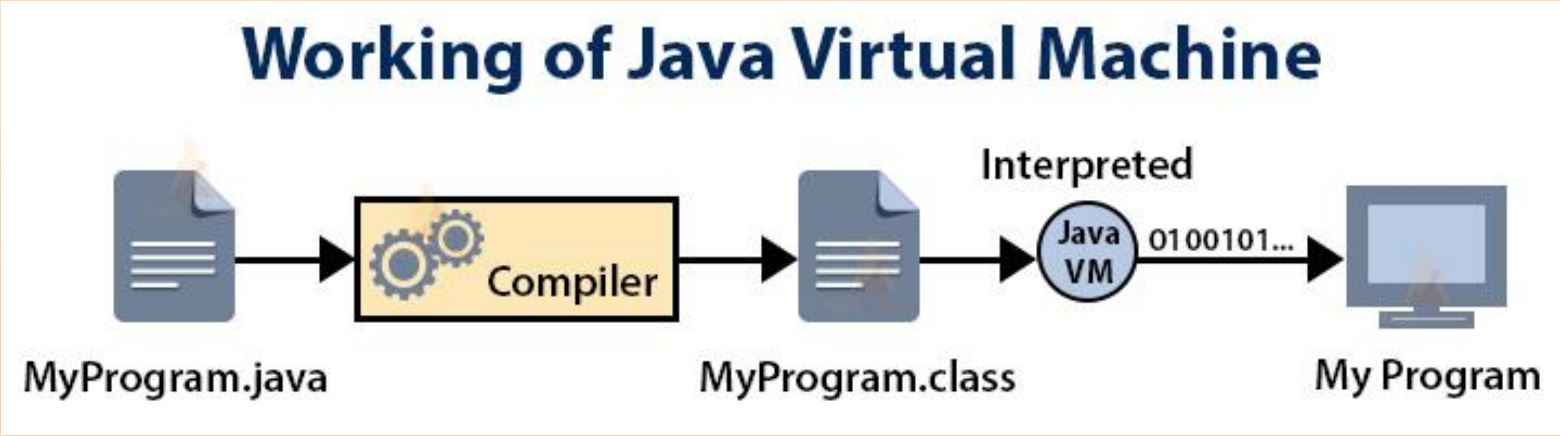
Object Oriented Programming Concept

- Java is Object Oriented because of the following facts: In Object Oriented concept each and everything is considered as an object having states(data) and behaviors(actions)
- Actions are always performed on data.
- C++ which is semi-object-oriented whereas the Java is the fully object-oriented programming language.
- It has all OOP features such as Inheritance, Abstraction, Encapsulation, and Polymorphism. Here are the main concepts of OOP's.
  - **Class**
  - **Object**
  - **Inheritance**
  - **Polymorphism**
  - **Abstraction**
  - **Encapsulation**

Features Of java

Compiled & Interpreted

- Some programming languages are either compiled or interpreted whereas Java is both compiled or interpreted.
- Java compiler translates java source code to the bytecodes.
- Java interpreter executes the translated bytecodes directly on the system that supports the Java Virtual Machine.
- Bytecode is interpreted on any platform by JVM.



Features Of java

Robust

- Java is robust as it is capable of **handling run-time errors**, supports automatic garbage collection and exception handling, and avoids explicit pointer concept.
- Java has a strong memory management system. It helps in eliminating errors as it checks the code during both compile and runtime.
- Java is *garbage-collected language* – JVM automatically deallocates the memory blocks and programmers do not have to worry about deleting the memory manually as in case of C/C++.
- Java also provides the concept of exception handling which identifies runtime errors and eliminates them.
- In Java, any runtime error encountered by the JVM is never passed directly to the underlying system rather immediately **terminates the program** stopping it from causing any harm to the underlying system.

Features Of java

Architecture Neutral

- Architectural neutral which can run on any available processors.
- To execute Java application anywhere on the network which composed of the variety of CPU and operating system architectures.
- The compiler generates an architecture-neutral object file format.

Dynamic

- Java is Dynamic because of bytecode.
- Java loads its all classes, jars, and libraries from hard drive during runtime and also Java support dynamic binding.
- Overriding due to these above two major feature Java is considered as dynamic.
- Allocates memory at runtime using the new operator.

Features Of java

Distributed

- Java is distributed because it encourages users to create distributed applications.
- In Java, we can split a program into many parts and store these parts on different computers. A Java programmer sitting on a machine can access another program running on the other machine.
- This feature in Java gives the advantage of distributed programming, which is very helpful when we develop large projects. Java helps us to achieve this by providing the concept of **RMI (Remote Method Invocation)** and **EJB (Enterprise JavaBeans)**.
- Java comes with an extensive library of classes for interacting, using TCP/IP protocols such as HTTP and FTP, which makes creating network connections much easier than in C/C++.
- It also enables multiple programmers at many locations to work together on a single project.

Features Of java

High Performance

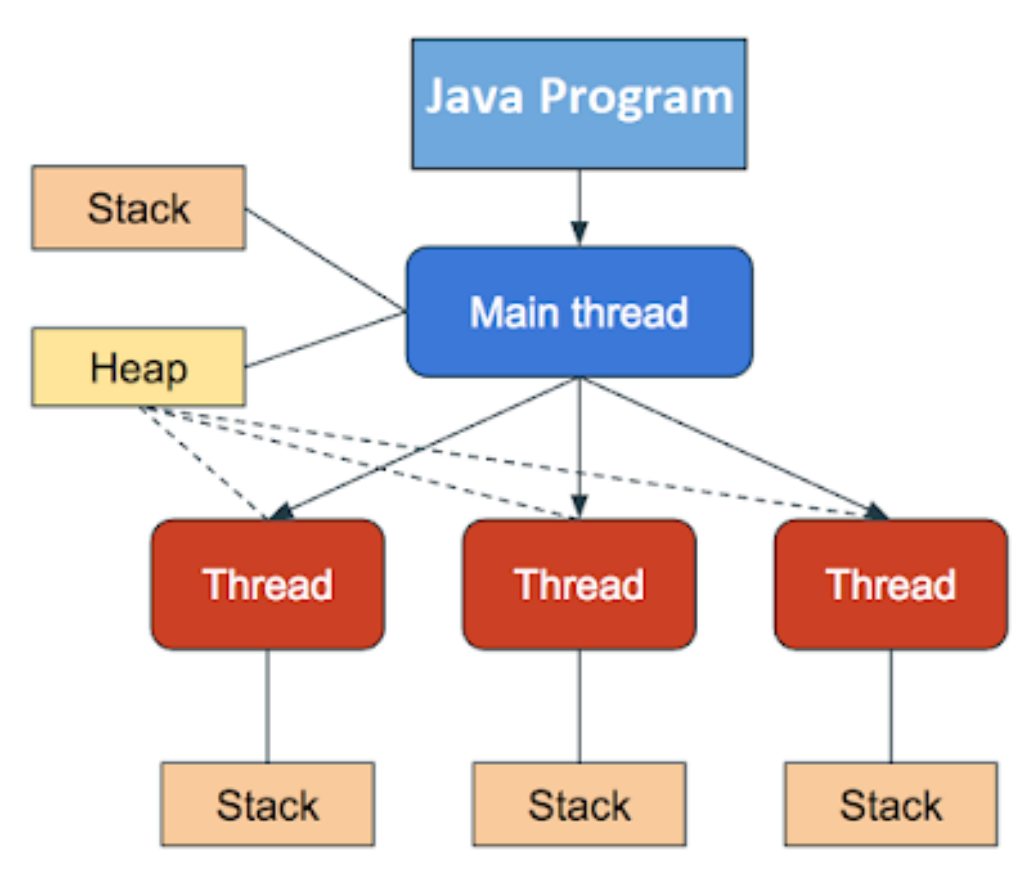
- Java has high performance due to a just-in-time compiler(JIT).
- Using Bytecode which is faster than ordinary pointer code.
- Garbage collector which removes the unused space automatically.
- The dynamic binding which is known as overriding due these above two major features of Java is considered as dynamic.
- Allocates memory at run-time using the new operator.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications etc.

Features Of java

Multithreading in Java is a process of executing multiple threads simultaneously. A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.



Difference between C++ and Java

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
Goto	C++ supports goto statement.	Java doesn't support goto statement.
Multiple inheritance	C++ supports multiple inheritances.	Java doesn't support multiple inheritances through class. It can be achieved by interfaces in java.

Difference between C++ and Java

Comparison Index	C++	Java
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write pointer program in C++.	Java supports pointer internally. But you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only.	Java uses compiler and interpreter both.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.

Difference between C++ and Java

Comparison Index	C++	Java
Inheritance Tree	C++ creates a new inheritance tree always.	Java uses single inheritance tree always because all classes are the child of Object class in java. Object class is the root of inheritance tree in java.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comment.	Java supports documentation comment (/** ... */) to create documentation for java source code.

Difference between C++ and Java

Comparison Index	C++	Java
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.

Java Versions

Java SE Version Version Number & Release Date	
JDK 1.0 (Oak) January 1996	
JDK 1.1 February 1997	Inner Class, JavaBeans, JDBC, RMI, AWT event model, Reflection, JIT(Just In Time) compiler on Microsoft Windows platforms,
J2SE 1.2 (Playground) December 1998	Java plug-in, Java IDL, an IDL implementation for CORBA interoperability Collections framework, the Swing graphical API was integrated into the core classes Sun's JVM was equipped with a JIT compiler for the first time
J2SE 1.3 (Kestrel) May 2000	HotSpot JVM included, RMI was modified to support optional compatibility with CORBA, JNDI (Java Naming and Directory Interface), Java Platform Debugger Architecture (JPDA) included, JavaSound, Synthetic proxy classes.

Java Versions

Java SE Version Version Number & Release Date	
JDK 1.0 (Oak) January 1996	
J2SE 1.4 (Merlin) 1.4 February 2002	Improved libraries, Perl regular expressions included, Provided exception chaining (It allows an exception to encapsulate original lower-level exception), IPv6 support (Internet Protocol version 6). Logging API (Specified in JSR 47, Image I/O API for reading and writing images in formats like JPEG and PNG, XML parser and XSLT processor integrated, Security and cryptography extensions (JCE, JSSE, JAAS) integrated.
J2SE 5.0 (Tiger) September 2004	It provided compile-time (static) type safety for collections and eliminates the need for most typecasts, Used Metadata or annotations, Autoboxing/unboxing, Enumerations, Enhanced for each loop, Improved semantics of execution for multi-threaded Java programs, Static imports. There were also some improvements in standard libraries: Automatic stub generation for RMI objects, Swing: It provided a skinny look and feel, The concurrency utilities in package java.util.concurrent, Scanner class for parsing data from various input streams and buffers.

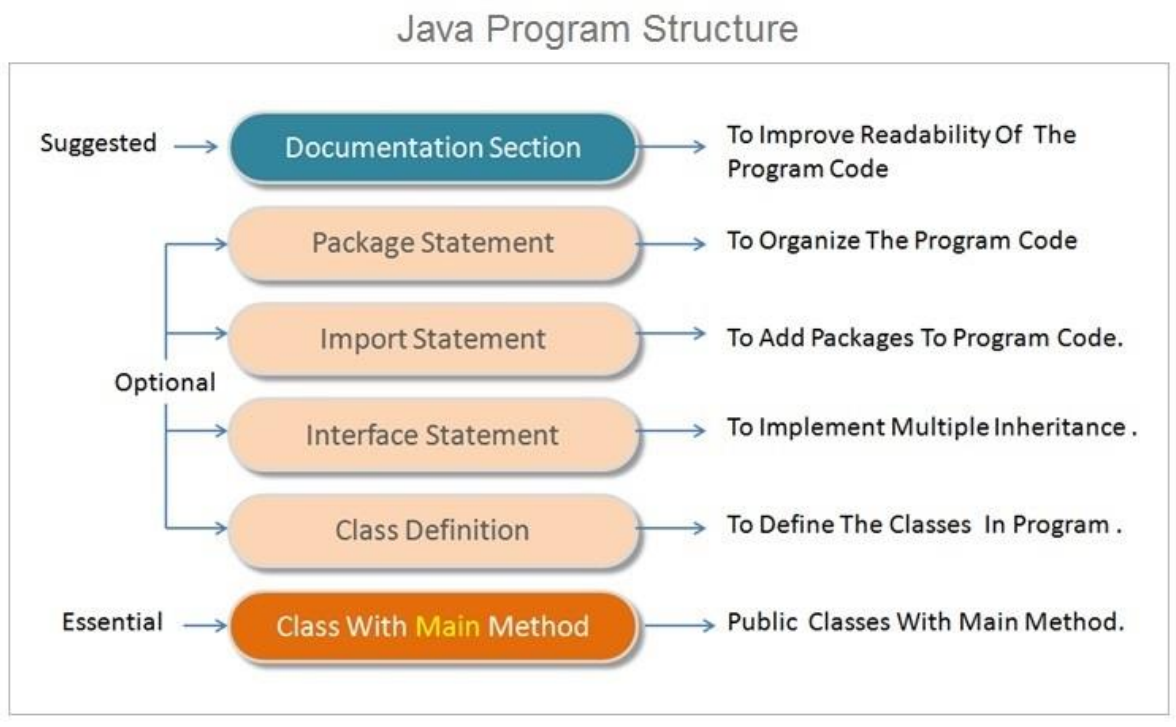
Java Versions

Java SE Version Version Number & Release Date	
Java SE 6 (Mustang) December 2006	Dropped the support for older Win9x versions, Scripting Language Support, Generic API for tight integration with scripting languages, Improved Web Service support, JDBC 4.0 support. Use a Java Compiler API to invoke a Java Compiler programmatically.
Java SE 7 (Dolphin) July 2011	JVM support for dynamic languages, Compressed 64-bits pointer, Strings added in switch, Automatic resource management in try-statement, Underscores allowed in numeric literals, Binary integer literals. Improved type interface for creating generic instance. (also called diamond operator <>), Improved catching and throwing. (catch multiple exceptions and rethrow with improved type checking), Provided Java Deployment rulesets.
Java SE 8 March 2014	Language-level support for Lambda expressions. Allowed developers to embed JavaScript code within applications, Annotation of Java Types, Provided Date and Time API, Repeating Annotations, Launching of JavaFX applications, Removal of permanent generation.
Java SE 9 September, 2017	Modularization of the JDK under Project Jigsaw, Provided Money and Currency API, Tight integration with JavaFX, Java implementation of reactive streams, More Concurrency Updates, Provided Java Linker, Automatic scaling and sizing.

Java Versions

Java SE Version Version Number & Release Date	
Java SE 10 March, 20th 2018	Local-Variable Type Inference, Experimental Java-Based JIT Compiler This is the integration of the Graal dynamic compiler for the Linux x64 platform, Application Class-Data Sharing This allows application classes to be placed in the shared archive to reduce startup and footprint for Java applications Time-Based Release Versioning Parallel Full GC for G1 Garbage-Collector Interface Additional Unicode Language-Tag Extensions Root Certificates Thread-Local Handshakes Heap Allocation on Alternative Memory Devices Remove the Native-Header Generation Tool - javah Consolidate the JDK Forest into a Single Repository
Java SE 11 September, 25th 2018	
Java SE 12 March, 19th 2019	
Java SE 13 September, 17th 2019	
Java SE 14 March, 17th 2020	
Java SE 15 *Expected in September 2020*	

Java Program Structure



Documentation section

It is a set of comment lines explaining the program's purpose and improves the readability of the program. It is a non-executable statement that helps to understand a program especially when your programs get more complex. It exists only for the programmer and is ignored by the compiler. A good program should contain comments that describe the information which the programmer would like to refer e.g. purpose of the program, author name, date and time of program creation. Such comments begin with delimiter /** and end with */.

Java Program Structure

Package declaration

Java allows you to group classes in a collection known as package. This statement declares a package name and informs the compiler that the classes defined here belong to this package. The package declaration consists of the keyword package, a space, and the name of the package . It must appear as the first statement in the source code before any class or interface declaration. Only one package declaration can appear in the source file. Package declaration is optional.

Example:

```
package staff;
```

Java Program Structure

Import statements

It is Similar to #include statement in C. In java, we have lot of predefined classes that are stored into packages. To refer these predefined classes, you have to use fully qualified name like packagename.classname. Import statements are used to access the classes and interfaces that are part of other packages. An import statement tells the Java compiler, this Java file is using which other Java files contained in that package. It needs to retype the package path name along with the classname. We can omit this by using import keyword.

Example

import java.util.Scanner; //imports only the Scanner class from java.util package.

This statement declares that all classes and interfaces defined in this source file are part of the staff package.

Java Program Structure

Interface statement

It is similar to a class but only includes constants and group of method declaration. It is used in java, when we want to implement multiple inheritance feature. Interfaces cannot be instantiated. They can only be implemented by classes or extended by other interfaces. It is an optional section.

Example

```
interface ExampleInterface
{
        public void method1();
   // All the methods are public abstract by default public void method2();
  //Note down that these methods are not having body
}
```

Java Program Structure

Class definition

It describes information about user defined classes of the program. Classes are primary feature of Java program. Every Java program consists of at least one class definition with main method which is starting point of program execution and follows the statements in the order specified. A class is a collection of data members(variables) and methods that operate on the fields. Java program may contain multiple class definition. They are used to map real world problems. Java program may contain multiple class definition. Class is the compulsory section.

Java Program Structure

Main Method Class

It is the starting point of the program. The main method can have the ability to create objects, evaluate expressions, and invoke other methods. On reaching the end of main, the program terminates. and the control passes back to the operating system. Java Compiler will compile class that do not contain main() method. If main() is not declared, it reports runtime error, because at compile time Compiler is not using main() but interpreter use it at runtime.

Example for Java Structure

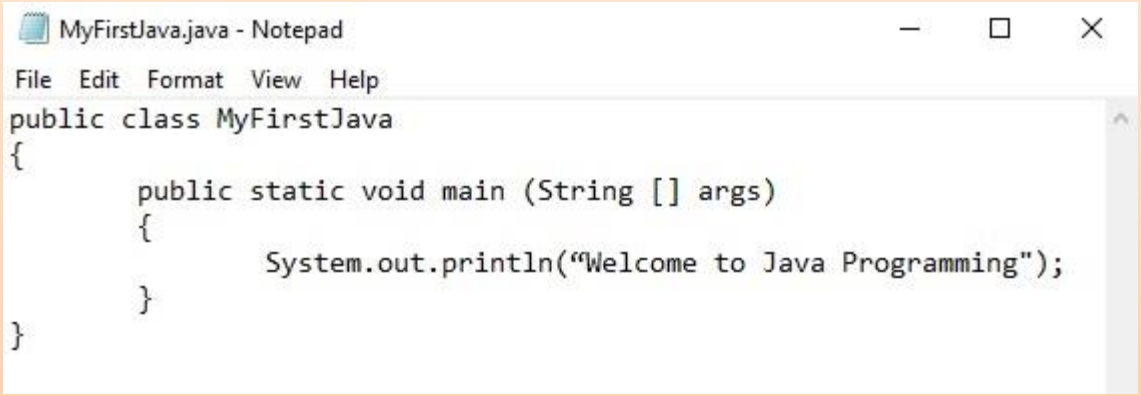
```
public class MyFirstJava
{
    public static void main(String[] args)
    {
        System.out.println(“Welcome to Java Programming");
    }
}
```

Java Program Structure

#1) Open notepad and type the following code.

```
public class MyFirstJava
{
    public static void main (String [] args)
    {
        System.out.println("Welcome to Java Programming");
    }
}
```

#2) Save the above file as "MyFirstJava.java" in your code directory.

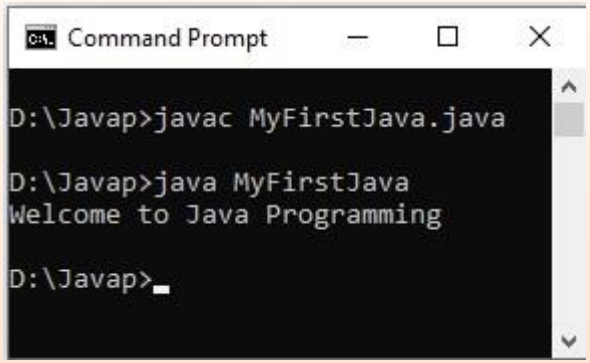


As the class "MyFirstJava" is declared public, you will name the Java file with this class name. This is a Java convention.

#3) Open the command prompt and go to the directory where you just saved your Java file.

#4) Next type in the command "javac MyFirstJava.java" in the command prompt. The following output is displayed. Note that there are no errors and the file is

Java Program Structure



Let us understand the program by listing the main Java keywords that are used:

Public: This is a visibility specifier or access specifier that defines the visibility of the component. Public means the parameter or component is visible to all. In the above program, we use a "public" specifier twice i.e. first before the class definition and second for the main function. This means the class, as well as the main function, are public and visible to all.

Class: This is a Java keyword and is used to define a class. The "class" keyword is followed by a class name that defines the class. After the class name, a code block is defined with curly braces ({}). The code block will contain all the methods and data belonging to this class. Here we define a new class "MyFirstJava" using the class keyword.

Static: The keyword "static" is used to indicate that the method/object/variable that follows this keyword is static in nature and it can be invoked without using the object and the dot (.) operator. The static keyword before the main method means that the main method is static. The Main method is executed by JVM and by making

Java Program Structure

Void: The “void” keyword indicates that the method does not return anything.

Main: The keyword “main” that indicates the main method is the starting point of any Java program. The execution of a Java program begins with the main method.

String[] args: The string array args holds command line arguments.

System.out.println: System.out.println is used to print messages to the screen. The system is a class in Java. The parameters “Out” and “println” are the members of the PrintStream class. While “out” is an object, println is a method.

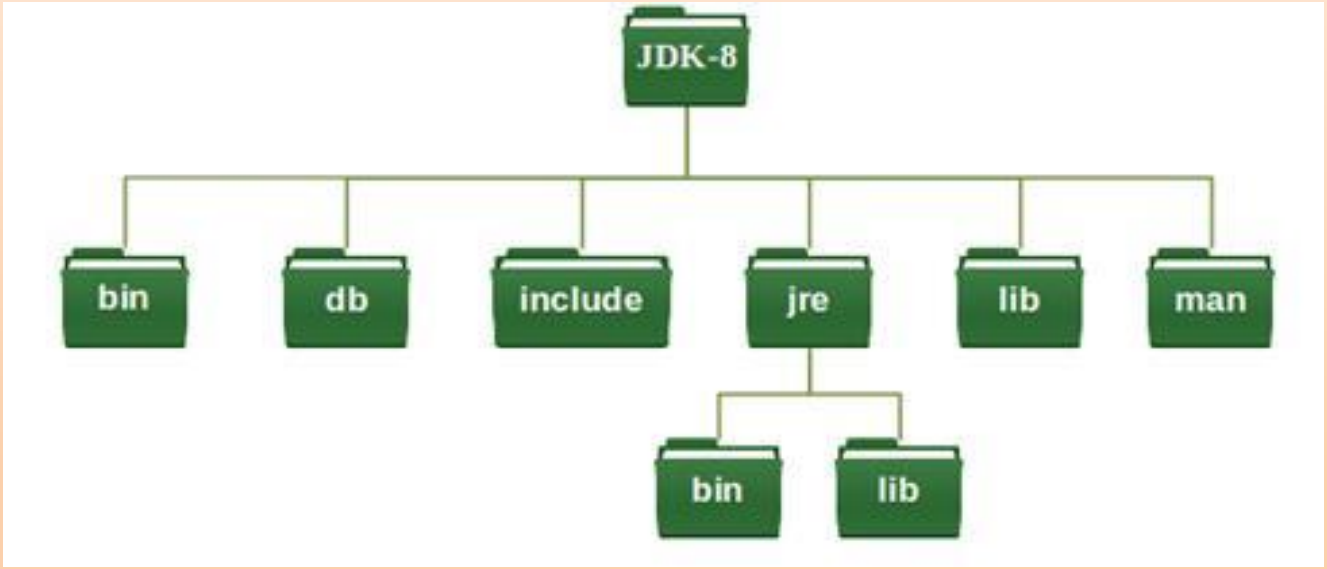
The following are the basic steps to create a simple program in Java.

1. Write the Java Source Code in any text editor.
2. Save the File with an extension “.java”.
3. Open a command prompt window.
4. Change the Directory to the one which has the Java program just created.
5. Compile your program using the “javac” command followed by the Java filename.
6. Execute the Program using the “java” command followed by the class name.

Java Directory Structure

The JDK Directory Structure

he JRE is the core platform of Java whereas the JDK includes the JRE along with development tools and libraries. The development tools and libraries aid in creating Java applications that run on JRE. One may install JRE or JDK according to the requirement. The first noticeable change can be seen the directory structure, as arranged by Java 9.
Prior to this version, Java had the following directory structure:



Java Directory Structure

The **JDK** directory is the root directory of the JDK software installation. It contains copyright, license, and README files. It also contains src.zip, the archive of source code for the Java platform.

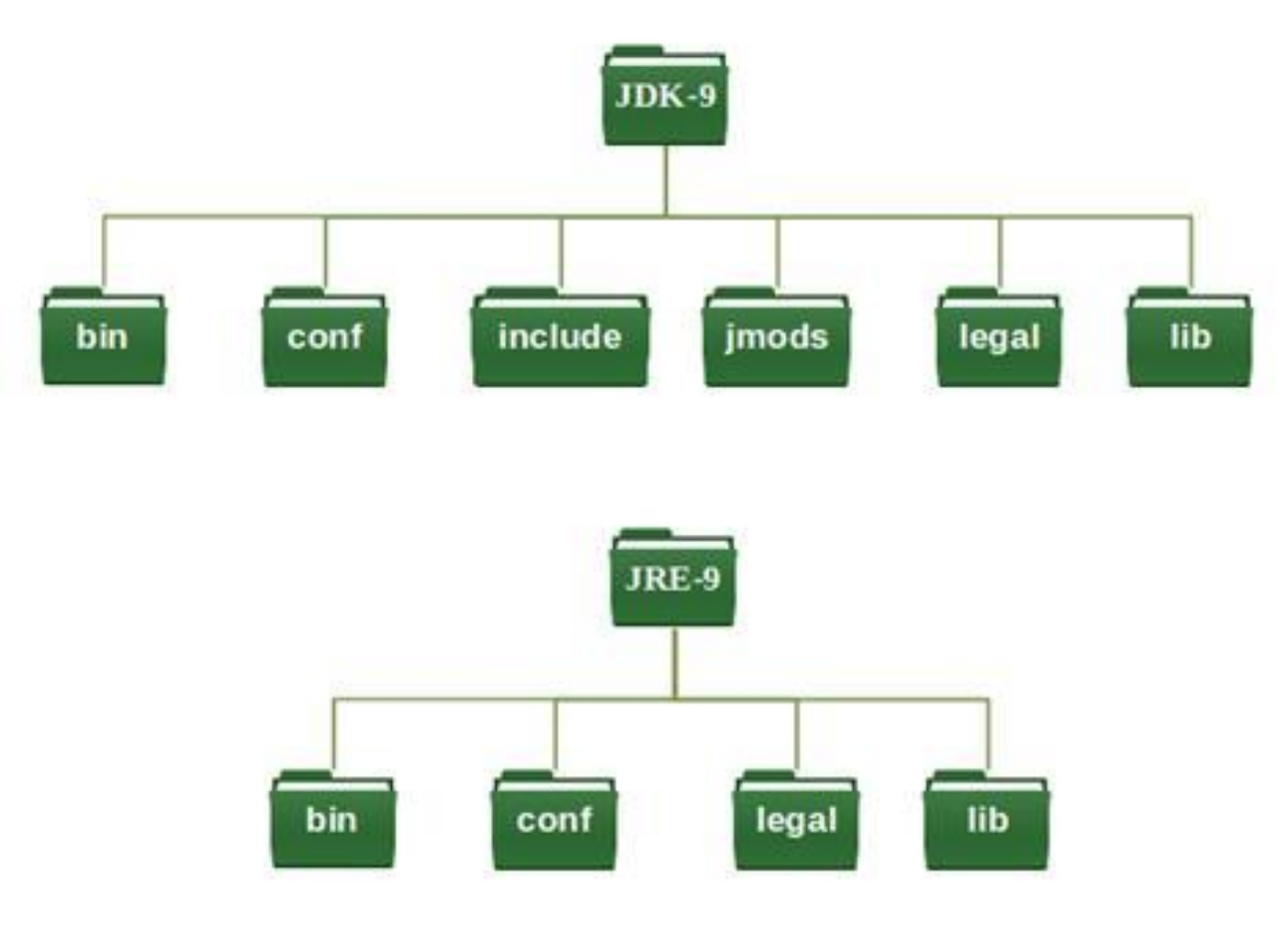
1. The **JDK/bin** directory contained the tools that were run from the command line, such as the *javac*, *javap*, *jconsole*, *jar*, *javadoc*, and so on. These tools are used for compiling, debugging, archiving and, other development purposes. Typically, the PATH environment variable contains an entry for this directory.
2. The **JDK/db** directory contains the Java DB database.
3. The **JDK/include** directory contains C-language header files that support native-code programming with the Java Native Interface and the Java Virtual Machine (JVM) Debugger Interface.

Java Directory Structure

4. The **JDK/jre** directory is the root directory of the Java Runtime Environment (JRE) used by the JDK development tools.
    - The **JDK/jre/bin** directory contains executable files for tools and libraries used by the Java platform.
    - The **JDK/jre/lib** directory contains code libraries, property settings, and resource files used by the JRE.
5. The **JDK/lib** directory contained the files that were used by the development tools, such as tools.jar, which contains non-core classes for support of the tools and utilities in the JDK.
6. The **JDK/man** directory contains man pages for the JDK tools.

Java Directory Structure

The new directory structure arranged by Java 9 is shown below:



Classes, Objects and Methods in Java

Classes, objects, and methods are fundamental concepts of Object-Oriented Programming (OOP) in Java. Understanding the relationship among them is essential before learning concepts such as inheritance, polymorphism, encapsulation, and abstraction.

Class

- A class is a blueprint or template for creating objects. It defines the data (fields/variables) and behaviour (methods) that objects of that class will have.

- A class is a user-defined data type that groups data members and methods into a single unit.

- A class itself is a conceptual design and **does not occupy memory**; memory is only allocated when an object (an instance of that class) is created.

Classes, Objects and Methods in Java

Components of a Java Class

A typical class can contain the following components:

- **Fields (Variables):** Represent the state or attributes of the class.
- **Methods (Functions):** Define the actions or behaviors the class can perform.
- **Constructors:** Special blocks of code used to initialize new objects.
- **Blocks & Nested Classes:** Internal code logic or helper classes defined inside the main class.

Real World Example

- Think of a class as an architectural **blueprint for a car**.
- The blueprint specifies that the car must have attributes like color and maxSpeed, and behaviors like accelerate() or brake().
- You cannot drive a blueprint. However, you can use that blueprint to build physical cars (Objects), like a red Tesla or a blue Ford.

Classes, Objects and Methods in Java

```java
// 1. Defining the Class Blueprint
class Car {
    // Fields (Attributes)
    String brand;
    String color;
    // Constructor to initialize the object
    Car(String brand, String color) {
        this.brand = brand;
        this.color = color;
    }
    // Method (Behavior)
    void drive() {
        System.out.println("The " + color + " " + brand + " is driving!");
    }
}
```

Classes, Objects and Methods in Java

```java
class Student {

    // Data member
    String name;
    int age;

    // Method
    void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

Here, Student is a **class**.
It contains:

- name → data member
- age → data member
- display() → method

The plan itself is not an actual student. It describes what a Student object will contain.

Classes, Objects and Methods in Java

Object

- An object is an instance of a class. It represents an actual entity created from the class and has its own state and behaviour.
- An object is an instance of a class that occupies memory and can access the data members and methods defined by the class.
- Each object holds its own state.
  - **State**: Values stored in fields.
  - **Behavior**: Actions defined through methods.
  - **Identity**: Distinguishes one object from another.

  Objects mirror real-world items such as customer, product or circle. Non-primitive objects are stored on the heap while their references remain on the stack.

Creating an Object

The new keyword is generally used to create an object.

```
Student s1 = new Student();
```

Classes, Objects and Methods in Java

```java
class Student {

    // Data member
    String name;
    int age;

    // Method
    void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
    public static void main(String[] args)
    {
        Student s1 = new Student();
        s1.name="Rahul";
        s1.age=25;
        s1.display();
    }
}
```

Classes, Objects and Methods in Java

```
class Vehicle{
    int numberOfWheels;
    String brandName, color;
    double price;
    void start( );
    void changeGear( );
}
```

blue print of an object

```
Vehicle car = new Vehicle( );
```

actual object

```
Vehicle bike = new Vehicle( );
```

```
Vehicle truck = new Vehicle( );
```

Classes, Objects and Methods in Java

```java
class Vehicle {
    // Data members
    int numberOfWheels;
    String brandName;
    String color;
    double price;
    // Method
    void start() {}
    void changeGear() {}
    void display() {}

    public static void main(String[] args) {

        // Creating Car object
        Vehicle car = new Vehicle();
        // Creating Bike object
        Vehicle bike = new Vehicle();
        // Creating Truck object
        Vehicle truck = new Vehicle();
    }
}
```

- **Class:** A blueprint or template that defines the properties (data members) and behaviours (methods) of objects.
- **Object:** An instance of a class created using the new keyword.
- **Data Members:** Variables that represent the properties/state of an object, such as brandName, color, and price.
- **Methods:** Functions that define the actions/behaviour of an object, such as start() and changeGear().
- **Multiple Objects:** A single class can be used to create multiple objects, and each object can have different values.

Classes, Objects and Methods in Java

- A method is a block of statements under a name that gets executes only when it is called. Every method is used to perform a specific task.

- The major advantage of methods is code re-usability (define the code once, and use it many times).

- In a java programming language, a method defined as a behavior of an object. That means, every method in java must belong to a class.

- Methods help us:
  - Reuse code
  - Divide a program into smaller modules
  - Reduce code duplication
  - Improve readability
  - Make programs easier to maintain

Classes, Objects and Methods in Java

- Every method in java must be declared inside a class.

- Every method declaration has the following characteristics.
  - **returnType** - Specifies the data type of a return value.
  - **name** - Specifies a unique name to identify it.
  - **parameters** - The data values it may accept or recieve.
  - **{ }** - Defienes the block belongs to the method.

Creating a method

A method is created inside the class and it may be created with any access specifier. However, specifying access specifier is optional.

Classes, Objects and Methods in Java

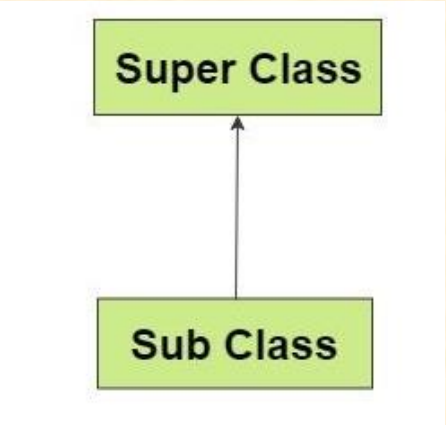
Following is the syntax for creating methods in java.

```
class <ClassName>
{
        <accessSpecifier> <returnType> <methodName>( parameters )
        {
                ... block of statements; ...
        }
}
```

- The methodName must begin with an alphabet, and the Lower-case letter is preferred.
- The methodName must follow all naming rules.
- If you don't want to pass parameters, we ignore it.
- If a method defined with return type other than void, it must contain the return statement, otherwise, it may be ignored.

Inheritance

- It is an important part of OOPs (Object Oriented programming system).
- It is the mechanism in java by which one class is allowed to inherit the features(fields and methods) of another class.
- It helps in creating a new class from an existing class, promoting code reusability and better organization.
  - A subclass can reuse the fields and methods of the parent class without rewriting the code
  - A subclass can add its own fields and methods or modify existing ones to extend functionality.



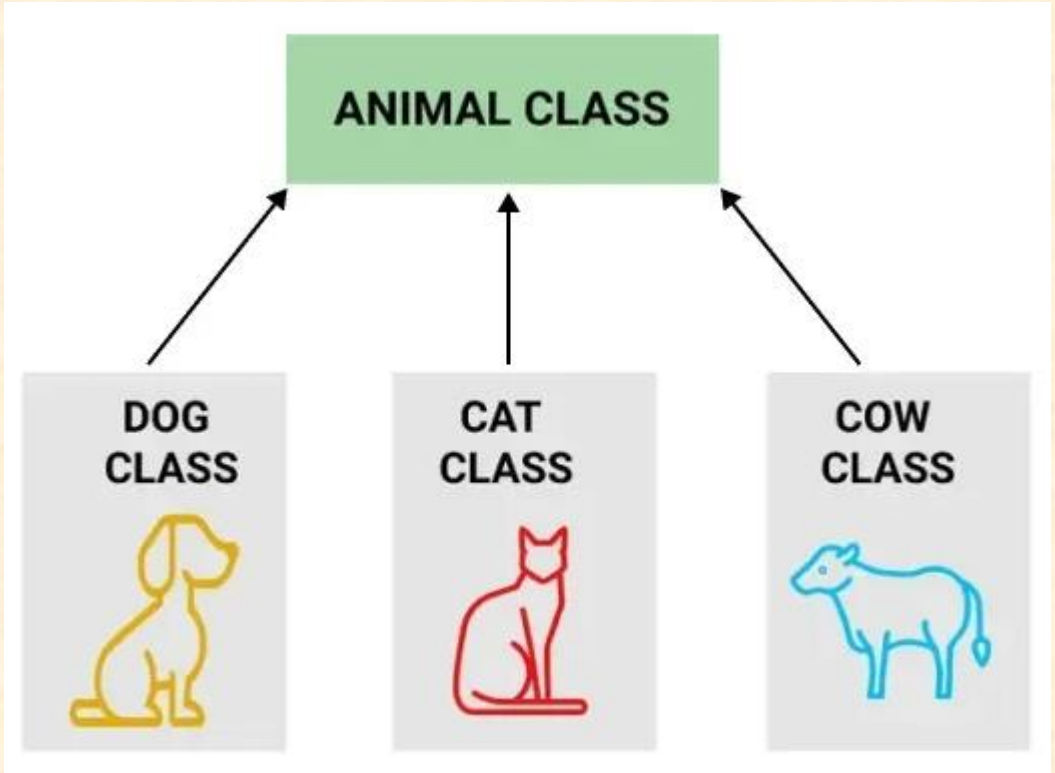
Inheritance

In Java, inheritance is implemented using the extends keyword.

```
class Parent {
// fields and methods
}
class Child extends Parent {
// additional fields and methods
}
```

Example

- Animal is the base class.
- Dog, Cat and Cow are derived classes that extend Animal class and provide specific implementations of the base/own method.



Inheritance

```java
// Parent class
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
// Child class
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
// Child class
class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows");
    }
}
```

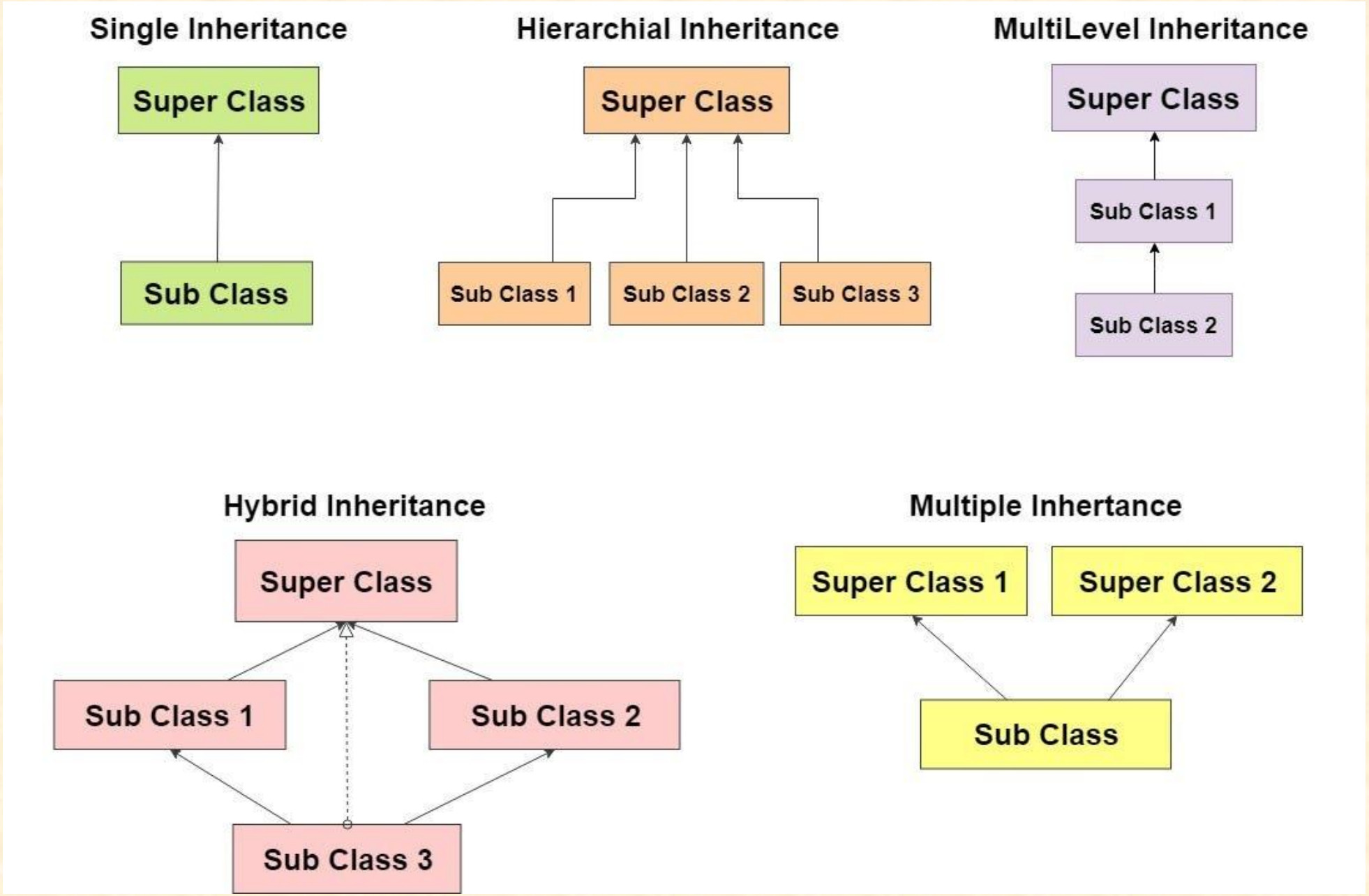
```java
// Child class
class Cow extends Animal {
    void sound() {
        System.out.println("Cow moos");
    }
}
// Main class
public class AnimalDemo {
    public static void main(String[] args) {
        Animal a;
        a = new Dog();
        a.sound();
        a = new Cat();
        a.sound();
        a = new Cow();
        a.sound();
    }
}
```

Inheritance

Why use inheritance in java?

- **Code Reusability:** Reuses the properties and methods of the superclass, reducing code duplication.
- **Abstraction:** Supports abstract classes and promotes simplified, maintainable, and extensible code.
- **Class Hierarchy:** Enables the creation of hierarchical relationships among related classes.
- **Polymorphism:** Allows subclasses to override superclass methods and provide different behaviors.

Types of Inheritance



Single Inheritance

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes, it is also known as simple inheritance.

```
//Super class
class Vehicle {
    Vehicle() {
        System.out.println("This is a Vehicle");
    }
}
// Subclass
class Car extends Vehicle {
    Car() {
        System.out.println("This Vehicle is Car");
    }
}
public class SingleInher {
    public static void main(String[] args) {
        // Creating object of subclass invokes base class constructor
        Car obj = new Car();
    }
}
```

Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also acts as the base class for other classes.

```
class Vehicle {
    Vehicle() {
        System.out.println("This is a Vehicle");
    }
}
class FourWheeler extends Vehicle {
    FourWheeler() {
        System.out.println("4 Wheeler Vehicles");
    }
}
class Car extends FourWheeler {
    Car() {
        System.out.println("This 4 Wheeler Vehicle is a Car");
    }
}
public class MultiLevelInher {
    public static void main(String[] args) {
        Car obj = new Car(); // Triggers all constructors in order
    }
}
```

Hierarchical Inheritance

In hierarchical inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class. For example, cars and buses both are vehicle

```
class Vehicle {
    Vehicle() {
        System.out.println("This is a Vehicle");
    }
}
class Car extends Vehicle {
    Car() {
        System.out.println("This Vehicle is Car");
    }
}
class Bus extends Vehicle {
    Bus() {
        System.out.println("This Vehicle is Bus");
    }
}
public class HierarchicalInher {
    public static void main(String[] args) {
        Car obj1 = new Car();
        Bus obj2 = new Bus();
    }
}
```

Multiple Inheritance

- Multiple Inheritance is a feature of object-oriented programming
- Where a child class can inherit properties of more than one parent class.
- The problem occurs when methods with the same signature or feature exist in both the parent classes.
- On calling the method, the compiler cannot determine which parent class method to call and even on calling which class method gets the priority.
- In that case, we use the **interface** in multiple inheritances.

Multiple Inheritance

```
interface LandVehicle {
    default void landInfo() {
        System.out.println("This is a LandVehicle");
    }
}
interface WaterVehicle {
    default void waterInfo() {
        System.out.println("This is a WaterVehicle");
    }
}
// Subclass implementing both interfaces
class AmphibiousVehicle implements LandVehicle, WaterVehicle {
    AmphibiousVehicle() {
        System.out.println("This is an AmphibiousVehicle");
    }
}
public class MultipleInher {
    public static void main(String[] args) {
        AmphibiousVehicle obj = new AmphibiousVehicle();
        obj.waterInfo();
        obj.landInfo();
    }
}
```

Multiple Inheritance

```java
// First interface
interface Camera {
    void takePhoto();
}

// Second interface
interface MusicPlayer {
    void playMusic();
}
// Concrete class implementing both interfaces
class SmartPhone implements Camera, MusicPlayer {
        @Override
    public void takePhoto() {
        System.out.println("Taking a photo...");
    }
    @Override
    public void playMusic() {
        System.out.println("Playing music...");
    }
}
```

```java
// Main class to run the code
public class Multiple1 {
    public static void main(String[] args) {
        SmartPhone myPhone = new SmartPhone();

        // Calling methods inherited from both interfaces
        myPhone.takePhoto();
        myPhone.playMusic();
    }
}
```