Java Overview and History

Java Editions

Java is categorized into three major editions, each serving a specific purpose:

Edition	Full Form	Purpose
Java SE (J2SE)	Java 2 Standard Edition	Used for standalone applications, provides fundamental Java features.
Java EE (J2EE)	Java 2 Enterprise Edition	Used for server-side applications, distributed computing, and web applications.
Java ME (J2ME)	Java 2 Micro Edition	Used for mobile and embedded systems programming.

1. Java SE (J2SE) - Java 2 Standard Edition

Java SE provides the core functionality of the Java programming language and is mainly used for standalone applications.

Key Features:

- 1. Basic Java Fundamentals: Includes essential features like Object-Oriented Programming (OOP), exception handling, multithreading, collections, etc.
- 2. Standalone Applications: Applications that run independently without a client-server architecture.
- 3. Examples of Standalone Applications: Calculator, Paintbrush, Calendar.
- 4. Usage Statistics: Direct utilization: 5%, Dependency on advanced technologies: 1000% (used in Selenium, Hadoop, Android, SAP, etc.)

2. Java EE (J2EE) - Java 2 Enterprise Edition

Java EE is used for creating large-scale enterprise applications that require a client-server model.

Key Features:

- 1. Server-Side Programming: Supports Servlets, JSP, EJB, and frameworks like Spring and Hibernate.
- 2. Distributed Applications: Applications that function over multiple machines.

- 3. Examples of Enterprise Applications: Banking Applications, Social Networking Platforms, E-commerce Websites, etc.
- 4. Usage Statistics: 95% of applications are distributed applications.

3. Java ME (J2ME) - Java 2 Micro Edition

Java ME is designed for mobile and embedded systems development.

Key Features:

- 1. Micro Programming: Optimized for resource-constrained environments.
- 2. Mobile-Based Applications: Applications tailored for mobile hardware architecture.
- 3. Examples of Mobile-Based Applications: Early mobile games, Feature phone applications.
- 4. Usage Statistics: 100% of mobile applications use mobile-based technology, but J2ME is no longer in high demand due to modern alternatives like Android.

History of Java

Java was created by Sun Microsystems in 1991 for embedded systems but later evolved into a general-purpose language.

Development Timeline:

Year	Event
1991	Sun Microsystems initiated a project for consumer electronic devices.
1992	A new programming language was created under Project 'Green'.
1992	James Gosling suggested the name 'OAK,' but it was already in use.
1994	The team adopted the name 'Java,' inspired by the coffee they drank.
1995	Sun Microsystems developed the first successful browser, 'Hot Java'.
1996	Java was officially introduced to the public.

Java Versions Timeline

Version	Year	Company
JDK 1.0	1996	Sun Microsystems

JDK 1.1	1997	Sun Microsystems
JDK 1.2	1998	Sun Microsystems
JDK 1.6	2006	Sun Microsystems
JDK 1.7	2011	Oracle Corporation
JDK 1.8	2014	Oracle Corporation
JDK 1.11	2018	Oracle (First licensed version)
JDK 1.22	2024	Oracle Corporation

Conclusion

Java has evolved from a simple language for electronic devices to a powerful, multi-platform programming language used in enterprise applications, web development, and mobile applications. It remains a critical tool in modern software development.

Differences Between Java and Other Languages

1. Java is a Dynamic Programming Language, whereas C and C++ are Static Programming Languages

Static Memory Allocation (C, C++)

- In C and C++, memory for primitive data types is allocated at **compile-time**.
- These languages follow **static memory allocation**, making them static programming languages.

Example:

```
int a = 10;  // Memory allocated at compile-time
```

Dynamic Memory Allocation (Java, Python)

- In Java, memory is allocated at **runtime** for primitive data types.
- Java follows **dynamic memory allocation**, making it a dynamic programming language.

Example:

```
int a = new Integer(10); // Memory allocated at runtime
```

Comparison Table:

Feature	C / C++	Java
Memory Allocation	**Static (Compile-time)**	**Dynamic (Runtime)**
Example	`int a = 10;`	`int a = new Integer(10);`
Programming Type	**Static Language**	**Dynamic Language**

---

2. Preprocessor is Required in C and C++, but Not in Java

C and C++:

- **Preprocessor:** The first phase of compilation in C and C++.
- The predefined libraries in C and C++ are provided as **header files (.h files).**
- These header files are included using **#include<>** statements.

Example:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
```

Preprocessor Actions in C and C++:

1. Recognizes `#include<>` statements.
2. Checks if the specified **header files exist** in C/C++ software.
3. If a header file is **missing**, it throws an **error**.
4. If the header file **exists**, its contents are **loaded into memory**.
5. **Static Loading:** Libraries are loaded **at compile-time**.

Java:

- **No Preprocessor:** Java does not have `#include<>` statements or **header files**.
- Java's predefined libraries are in the form of **classes and interfaces inside packages**.
- Packages are included using the **import** statement.

Example:

```
import java.io.*;
import java.util.*;
import java.sql.*;
```

Compiler Actions in Java:

1. Recognizes `import` statements.
2. Checks if the specified **packages exist** in Java software.
3. If a package **does not exist**, it throws an **error** (`package xxx does not exist`).
4. If the package **exists**, **no immediate loading** occurs.
5. At runtime, JVM **dynamically loads** the required classes and interfaces.
6. **Dynamic Loading:** Libraries are loaded **at runtime**.

Comparison Table:

Feature	C / C++ (Preprocessor)	Java (Compiler & JVM)
Library Format	**Header Files (.h)**	**Packages (Classes & Interfaces)**
Library Inclusion	Uses `#include<>`	Uses `import`
Processing Phase	Handled by **Preprocessor**	Handled by **Compiler & JVM**
Loading Type	**Static Loading** (Compile-time)	**Dynamic Loading** (Runtime)

---

3. Differences Between `#include<>` and `import` Statements

Feature	`#include<>` (C, C++)	`import` (Java)
Existence	Exists in C and C++	Exists in Java
Purpose	Includes header files	Includes classes and interfaces from packages
Evaluator	Handled by **Preprocessor**	Handled by **Compiler & JVM**
Loading Type	**Static Loading** (at compile-time)	**Dynamic Loading** (at runtime)
Multiple File Inclusion	Requires multiple `#include<>` statements for different header files	A single `import` statement can include multiple classes/interfaces from a package

---

Example Comparison:

C and C++ (`#include<>` - Multiple Statements Needed)

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
```

Java (`import` - Single Statement for Multiple Classes)

```
import java.io.*;
import java.util.*;
```

Conclusion

Feature	C / C++	Java
Memory Allocation	**Static (Compile-time)**	**Dynamic (Runtime)**
Preprocessor Requirement	**Yes**	**No**
Library Format	**Header Files (`.h`)**	**Packages (Classes & Interfaces)**
Loading Type	**Static Loading**	**Dynamic Loading**

Key Takeaways

✓ **Java dynamically allocates memory at runtime**, making it more flexible than C/C++. ✓ **Java does not need a preprocessor** since it uses `import` statements instead of `#include<>`. ✓ **Java supports dynamic loading**, reducing memory usage at compile-time. ✓ **Java's package-based approach is more efficient** than C/C++'s header file system.

Differences Between Java and Other Programming Languages

3. Platform Dependency: Java vs. C and C++

Platform Dependent Languages (C, C++)

- If a language requires the **same operating system** for both **compilation** and **execution**, it is considered **Platform Dependent**.
- Example: **C and C++**.
- When compiling C/C++ programs on **Windows**, the compiler generates an **.exe file**.
- This **.exe file** contains **Windows-specific** instructions and can only run on **Windows**.

Illustration of Platform Dependency:

```
 Compilation (Windows)  --> .exe file (Windows format)  --> Execution
(Windows only)
```

Platform Independent Language (Java)

- If a language allows **compilation in one OS** and **execution in multiple OS**, it is **Platform Independent**.
- Example: **Java**.
- Java uses the principle **"Write Once, Run Anywhere" (WORA).**
- Java compiler generates a **.class file** containing **bytecode** (not specific to any OS).
- The **Java Virtual Machine (JVM)** converts this bytecode into the local OS format.

Illustration of Platform Independence:

```
 Compilation (Windows)  --> .class file (Bytecode)  --> Execution
(Windows, Linux, Mac) via JVM
```

Comparison Table: .exe vs .class File

Feature	.exe File (C/C++)	.class File (Java)
Language	C, C++	Java
Contains	Directly executable code	Intermediate bytecode
OS Dependency	Platform Dependent	Platform Independent
Security	Less secure	More secure

---

4. Pointer Variables: C/C++ vs. Java

Pointer Variables in C and C++

- A pointer stores the **memory address** of another variable.
- Used for **direct memory access** and **dynamic memory allocation**.

Example in C++:

```
int a = 10;
int *p = &a; // Pointer storing address of a
```

Why Java Does Not Support Pointers?

1. **Memory Allocation**: Pointers require **static memory allocation**, but Java follows **dynamic memory allocation**.
2. **Security**: Direct memory access can lead to **security vulnerabilities**.

3. **Platform Independence**: Pointers work well with **platform-dependent** languages, but Java is **platform-independent**.
4. **Simplicity**: Pointers increase **complexity**, making Java harder to learn.

Reference Variables in Java

- Instead of pointers, Java uses **reference variables** that store **object references**.
- Reference values are **unique identities (hashcodes)** assigned by the **Heap Manager**.

Example in Java:

```
class Test {
    int a = 10;
}
Test obj = new Test(); // Reference variable storing object reference
```

Comparison Table: Pointer vs. Reference Variables

Feature	Pointer (C/C++)	Reference (Java)
Existence	Present in C/C++	Present in Java
Stores	Memory address of a variable	Object reference
Memory Allocation	Static	Dynamic
Platform Dependency	Platform Dependent	Platform Independent
Security	Less secure	More secure

---

5. Multiple Inheritance: Java vs. C++ and Python

What is Inheritance?

- **Inheritance** allows one class (child class) to acquire properties and methods from another class (parent class).
- This helps in **code reusability** and **modularity**.

Types of Inheritance

1. **Single Inheritance** - One parent class, one child class.
2. **Multiple Inheritance** - One child class inheriting from multiple parent classes.

Java Supports Single Inheritance

```
class Parent {
    int a = 10;
```

```
}
class Child extends Parent {
    int b = 20;
}
```

Why Java Does Not Support Multiple Inheritance?

- If two parent classes have the **same method or variable**, the compiler cannot decide which one to inherit.
- This creates an **ambiguity problem**.
- Java avoids this confusion to maintain **simplicity**.
- Java allows **Multiple Inheritance through Interfaces**, not classes.

Example of Multiple Inheritance in Python (Supported)

```
class A:
    i = 10
class B:
    i = 20
class C(A, B):
    pass
print(C.i)  # Output depends on the order of inheritance
```

Example of Multiple Inheritance in Java (Not Supported)

```
class A {
    int a = 10;
}
class B {
    int a = 20;
}
// class C extends A, B { } // INVALID in Java
```

Comparison Table: Single vs. Multiple Inheritance

Feature	Single Inheritance	Multiple Inheritance
Support in Java	✓ Yes	✗ No
Ambiguity	No ambiguity	Causes ambiguity
Implementation in Java	**Using `` keyword**	**Using Interfaces**

Conclusion

Feature	C / C++	Java
Platform Dependency	Dependent	Independent (JVM-based)
Memory Management	Static (pointers)	Dynamic (heap, reference variables)

Feature	C / C++	Java
Security	Less secure	More secure
Multiple Inheritance	Supported	Not supported (Only through interfaces)

Key Takeaways

✓ Java is **Platform Independent** because of **JVM**. ✓ Java **eliminates pointers** for security and simplicity. ✓ Java does **not support Multiple Inheritance** to avoid ambiguity. ✓ Java **achieves inheritance through Interfaces** instead.

End of Notes

Key Differences Between C++, Java in Object-Oriented Features

6. Destructors in C++ vs. Java

Concept of Object Creation and Destruction

In Object-Oriented Programming, data is represented in the form of objects. The lifecycle of an object involves:

1. **Creation of Objects** - Handled using **constructors**.
2. **Destruction of Objects** - Handled using **destructors** (except in Java where automatic garbage collection is used).

Destructors in C++

- In **C++**, developers are responsible for both creating and destroying objects.
- C++ **does not provide an automatic mechanism** for object destruction.
- **Explicit destructors** must be defined in the class to release resources such as memory, file handles, etc.
- Destructor syntax in C++:

```
class Example {
public:
    ~Example() { // Destructor
        cout << "Object is being deleted";
    }
};
```

Destructors in Java

- In **Java**, developers only create objects, but they do **not** explicitly destroy them.
- Java provides an **automatic memory management system** known as the **Garbage Collector (GC)**.
- The GC automatically deallocates memory occupied by objects **when they are no longer in use**.
- Java **does not support destructors** but provides a `finalize()` method (though it's discouraged in modern Java versions).
- Example (Not recommended in modern Java):
- ```
  class Example {
  ```
- ```
      protected void finalize() { // Finalize method (Not recommended)
  ```
- ```
          System.out.println("Object is being deleted");
  ```
- ```
      }
  ```
- ```
  }
  ```

Comparison Table

Feature	C++	Java
Object Creation	Using Constructors	Using Constructors
Object Destruction	Explicit Destructor Required	Automatic via Garbage Collector
Memory Management	Manual	Automatic
Destructor Support	Yes	No

---

7. Call By Value vs. Call By Reference in C, C++, and Java

Parameter Passing Mechanisms

Mechanism	Explanation
Call by Value	A copy of the actual value is passed to the function. Changes in the function do not affect the original variable.
Call by Reference	The memory address (reference) of the variable is passed. Changes in the function directly affect the original variable.

C and C++ Parameter Passing

- **Supports both Call by Value & Call by Reference.**
- Call by Reference is achieved using **pointers.**
- Example in C++ (Call by Reference using pointers):
  ```

```
void changeValue(int* p) {
    *p = 100;
}
int main() {
    int x = 10;
    changeValue(&x); // Pass address of x
    cout << x; // Outputs 100
}
```

Java Parameter Passing

- **Only Call by Value is supported.**
- Even when passing objects, Java **passes the reference value (memory address as a hashcode), but not the actual reference.**
- Example in Java:

```
class Example {
    int value = 10;
}
public class Test {
    static void modify(Example obj) {
        obj.value = 100;
    }
    public static void main(String[] args) {
        Example e = new Example();
        modify(e);
        System.out.println(e.value); // Outputs 100 (because Java passes the reference value)
    }
}
```

- **Even though objects appear to be modified, Java still follows Call by Value, because it passes the object's reference value (hashcode) rather than a direct reference.**

Comparison Table

Feature	C	C++	Java
Supports Call by Value	Yes	Yes	Yes
Supports Call by Reference	Yes (via pointers)	Yes (via references)	No
Object Passing	Not applicable	Call by Reference	Passes object reference value (Call by Value)

---

8. Operator Overloading in Java vs. C++

What is Operator Overloading?

- Operator overloading allows operators like +, -, *, etc., to be used with user-defined types.
- Example in C++:

```
class Complex {
public:
    int real, imag;
    Complex(int r, int i) { real = r; imag = i; }
    Complex operator+(Complex obj) { // Operator Overloading
        return Complex(real + obj.real, imag + obj.imag);
    }
};
```

Operator Overloading in Java

- **Java does not support operator overloading.**
- Reason: Java aims to maintain **code simplicity and readability**.
- Instead of operator overloading, Java uses **method overloading** to achieve similar results.
- Example using methods in Java:

```
class Complex {
    int real, imag;
    Complex(int r, int i) { real = r; imag = i; }
    Complex add(Complex obj) { // Method Overloading
        return new Complex(real + obj.real, imag + obj.imag);
    }
}
```

Comparison Table

Feature	C++	Java
Supports Operator Overloading	Yes	No
Example of Overloading	+, -, *, etc., can be overloaded	Uses methods instead
Reason	Provides flexibility	Simplicity, Avoids complexity

---

9. Memory Allocation for Primitive Data Types

C and C++ Memory Allocation

- Memory allocation for primitive data types **depends on the operating system**.

- Example:
  - Integer (`int`) → 2 bytes (on older systems), 4 bytes (on modern systems).
  - Character (`char`) → 1 byte.

Java Memory Allocation (Fixed Sizes)

- Java assigns **fixed memory sizes** for primitive data types, regardless of OS.
- Memory sizes:

Data Type	Size
byte	1 byte
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes
char	2 bytes
boolean	1 bit

Why Does Java Use 2 Bytes for Characters?

- C and C++ use **ASCII (1 byte per character)**.
- Java uses **UNICODE (2 bytes per character)** to support multiple languages (Hindi, Japanese, etc.).

What is Unicode?

- Unicode is a **universal character encoding standard**.
- Supports **multiple languages** for **internationalization (I18N)**.

Conclusion

Feature	C++	Java
Destructor Required	Yes	No (Garbage Collector used)

Feature	C++	Java
Call By Reference Supported	Yes	No (Uses Call by Value with reference variables)
Operator Overloading	Supported	Not Supported
Memory Allocation	OS-dependent	Fixed sizes

Java Features and Naming Conventions

1. Java Features

The Java programming language provides a set of features that define its nature and usability.

Key Features of Java

Feature	Description
Simple	Java avoids complex features like pointers and multiple inheritance and has a user-friendly syntax.
Object-Oriented	Java follows Object-Oriented Programming (OOP) principles like encapsulation, inheritance, and polymorphism.
Platform Independent	Java applications can be compiled on one OS and executed on another.
Architectural Neutral	Java code can run on different hardware architectures without modification.
Portable	Java applications can run on different platforms without changes.
Robust	Java has automatic memory management and strong exception handling mechanisms.
Dynamic	Java supports dynamic memory allocation at runtime.

Feature	Description
Secure	Java provides multiple levels of security, including implicit, web, and network security.
Multi-threaded	Java supports concurrent execution of multiple threads.
Distributed	Java provides J2EE for building distributed applications.
Interpretive	Java code is compiled into bytecode, which is interpreted by the JVM.
High Performance	Java improves performance using multithreading and JIT compilation.

1.1 Simple

Java is considered simple due to the following reasons:

- Requires less execution time and memory.
- Avoids complex features like pointers and multiple inheritance.
- Uses simplified syntax for better readability.

1.2 Object-Oriented

Java is an **Object-Oriented Programming Language (OOP)** that represents data as objects and follows OOP principles:

- **Class & Object** - Basic building blocks of OOP.
- **Encapsulation** - Wrapping data and methods into a single unit.
- **Abstraction** - Hiding implementation details.
- **Inheritance** - Allowing a class to acquire properties of another.
- **Polymorphism** - Ability to perform a single action in multiple ways.
- **Message Passing** - Objects interact by exchanging messages.

1.3 Platform Independent

Java achieves platform independence through **Bytecode**, which can be executed on any operating system using the Java Virtual Machine (JVM).

Compilation & Execution Flow:

```
Java Source Code (.java) → Compiler → Bytecode (.class) → JVM → Machine Code
```

1.4 Architectural Neutral

- Java does not depend on specific hardware architectures.
- Java programs can be executed on different architectures without modification.

1.5 Portable

Java applications run on any OS or hardware without changes due to its **Write Once, Run Anywhere (WORA)** principle.

1.6 Robust

Java ensures robustness through:

- **Automatic Memory Management** using the **Garbage Collector**.
- **Exception Handling** using predefined libraries to handle runtime errors.

1.7 Dynamic

- Supports **Dynamic Memory Allocation**.
- Loads classes dynamically during runtime.

1.8 Secure

Java provides security at three levels:

Security Type	Description
Implicit Security	Security Manager inside JVM ensures safety.
Web Security	Uses JAAS (Java Authentication and Authorization Service).
Network Security	Provides built-in encryption algorithms.

1.9 Multi-threaded

Java supports **Multithreading**, allowing parallel execution of multiple tasks.

Thread Model	Description
Single Thread Model	One thread executes the entire application sequentially.

Thread Model	Description
Multi-thread Model	Multiple threads execute different parts of the application simultaneously.

1.10 Distributed

Java supports **Distributed Computing** through the **J2EE module**, which allows applications to run across multiple machines.

Application Type	Description
Standalone Applications	Run without a client-server architecture.
Distributed Applications	Use client-server architecture and distribute logic over multiple machines.

1.11 Interpretive

- Java is **both compiled and interpreted**.
- The **compiler** translates Java code into bytecode.
- The **interpreter (JVM)** executes the bytecode.

1.12 High Performance

Java ensures high performance through:

- **Multithreading**
- **Just-In-Time (JIT) Compiler** for faster execution
- **Optimized libraries and algorithms**

---

2. Java Naming Conventions

Java is **case-sensitive** and follows specific naming conventions for different elements.

2.1 Class, Abstract Class, Interface, and Enum Names

- Start with an **uppercase** letter.
- Use **CamelCase** format.
- Example:
- `String, StringBuffer, InputStreamReader`

2.2 Variable Names

- Start with a **lowercase** letter.
- Use **CamelCase** format.
- Example:
- `int count;`
- `String userName;`

2.3 Method Names

- Start with a **lowercase** letter.
- Use **CamelCase** format.
- Example:
- `getData(), setName(), calculateArea()`

2.4 Constant Variable Names

- Use **uppercase** letters.
- Separate words with **underscores (_)**.
- Example:
- `public static final int MAX_VALUE;`
- `public static final double PI;`

2.5 Package Names

- Use **lowercase** letters.
- Avoid underscores.
- Example:
- `java.io, java.util, javax.servlet`

2.6 Summary of Naming Conventions

Element	Convention	Example
Class/Interface	Start with uppercase, CamelCase	`EmployeeDetails`
Variable	Start with lowercase, CamelCase	`employeeName`
Method	Start with lowercase, CamelCase	`calculateSalary()`
Constant	Uppercase, separated by _	`MAX_LIMIT`
Package	Lowercase	`java.lang`

Conclusion

Java is a powerful, feature-rich, and well-structured programming language with a strong emphasis on **portability, security, and performance**. Following **proper naming conventions** ensures **readable and maintainable** code. Understanding these concepts will help developers write efficient and high-quality Java applications.

Java Programming Format

Structure of a Java Application

A Java application consists of several components that need to be structured properly. The following sections describe the key components:

---

1. Comment Section

Definition:

- Comments provide descriptions about the code for developers' understanding.
- They are non-executable and do not affect program execution.
- Commonly used for documentation, debugging, and explaining code functionality.

Types of Comments:

Type	Syntax	Usage
Single Line Comment	`// description`	Used for short comments within a single line.
Multi Line Comment	`/* description */`	Used for longer explanations spanning multiple lines.
Documentation Comment	`/** description */`	Used to generate API documentation.

Examples:

Single Line Comment

```
class Math {
    void add(int fval, int sval) { // It performs Arithmetic Addition
        // Code here
    }
}
```

Multi-Line Comment

```
class Math {
    /*
    Name: add
    Purpose: To perform Arithmetic Addition operation
    Params: int fval, int sval
    Return Value: int
    */
    public int add(int fval, int sval) {
        return fval + sval;
    }
}
```

Documentation Comment

```
/**
 * This class represents an Employee.
 */
public class Employee {
    /** Employee Number */
    public int eno;
    /** Employee Name */
    public String ename;
    /** Employee Salary */
    public float esal;
}
```

Generating API Documentation

- Java provides the `javadoc` tool to generate API documentation from documentation comments.
- **Command:** `javadoc Employee.java`
- Output: It generates `.html` documentation files.

---

2. Annotations in Java

Why Use Annotations Instead of Comments?

Feature	Comments	Annotations
Availability	Removed at compilation	Retained in bytecode and runtime

Feature	Comments	Annotations
Use Cases	Readability	Debugging, Testing, Frameworks (e.g., Spring, Hibernate)
Tools Support	Not Processed	Processed using reflection

Annotations vs XML

Feature	XML Based	Annotation Based
Complexity	High	Low
Error Prone	FileNotFoundException	None
Performance	Requires parsing	Directly available in Java code

---

3. Package Section

Definition:

A **package** is a collection of related classes and interfaces.

Advantages:

Feature	Description
Modularity	Groups related classes together
Abstraction	Hides implementation details
Security	Prevents unauthorized access
Reusability	Allows easy reuse of code
Shareability	Facilitates sharing across projects

Types of Packages:

Type	Example
Predefined Packages	`java.io`, `java.util`, `java.sql`

Type	Example
User-Defined Packages	`package myPackage;`

Rules for Package Declaration:

1. The **package statement** must be the **first statement** in a Java file.
2. **Only one package declaration** per Java file is allowed.

---

4. Import Section

Purpose:

- To use classes and interfaces from other packages.

Syntax:

Syntax Type	Syntax	Description
Import all classes	`import packageName.*;`	Imports all classes and interfaces from the package.
Import specific class	`import packageName.ClassName;`	Imports a specific class from a package.

Example:

```
import java.io.*; // Imports all classes in java.io
import java.util.Scanner; // Imports only Scanner class
```

Fully Qualified Name (FQN) Usage:

If a class is not imported, it must be accessed using its full package name.

```
java.util.ArrayList list = new java.util.ArrayList();
```

---

5. Classes and Interfaces Section

Definition:

Classes and interfaces represent real-world entities in Java.

Example:

```
class Student {
```

```
    int id;
    String name;
}

interface Vehicle {
    void start();
}
```

---

6. Main Class and main() Method

Definition:

- A class containing the `main()` method is called the **Main Class**.
- The `main()` method serves as the **entry point** of a Java application.

Syntax:

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

Why is main() Required?

Feature	Description
Execution Start	JVM starts execution from `main()`
Program Termination	JVM exits when `main()` completes
Standard Convention	Recognized across Java programs

Conclusion

This document provides a structured format for Java applications covering comments, annotations, packages, import statements, classes, interfaces, and the main class. By following this structured approach, Java programs become more readable, maintainable, and scalable.