**JAVA-NOTES**

**Introduction to Java: Simple Overview**

Java is one of the most popular and widely used programming languages in the world. It has powered everything from web applications and mobile apps to large-scale enterprise systems. To understand Java's significance, let's delve into its history and development in a straight forward way.

**The Birth of Java**

* **Who created Java?** Java was developed by a team of engineers at Sun Microsystems, led by James Gosling, in the early 1990s.
* **Why was it created?** Initially, Java was designed for interactive television and other embedded devices. The idea was to create a platform-independent programming language that could run on any device without needing to be recompiled for each platform.

**Timeline of Java's Development**

1. **1991: The Beginning**
   * Java's development began under the project name "Green Project."
   * The language was initially called **Oak**, named after an oak tree outside James Gosling's office.
   * It was intended for devices like set-top boxes and TVs, but these markets were not yet ready for the technology.
2. **1995: Java is Born**
   * The language was renamed to **Java**, inspired by a type of coffee popular among the development team.
   * Java's "Write Once, Run Anywhere" (WORA) principle became its defining feature. It meant that Java programs could run on any device with a Java Virtual Machine (JVM), regardless of the underlying hardware or operating system.
3. **1996: Java 1.0**
   * Sun Microsystems officially released Java 1.0.
   * This version allowed web developers to add dynamic and interactive elements (like applets) to web pages.
4. **1997–2000: Rapid Adoption**
   * Java gained immense popularity for its simplicity and cross-platform capabilities.
   * Companies began adopting Java for server-side applications, especially in web and enterprise systems.
5. **2006: Open Source Transition**
   * Sun Microsystems made Java open-source, making it more accessible and accelerating its development through community contributions.
6. **2010: Oracle Acquires Java**
   * Oracle Corporation acquired Sun Microsystems, taking over the development and management of Java.
   * Under Oracle, Java received regular updates and improvements.

**Key Features of Java**

* **Platform Independence:** Java programs can run on any system with a JVM, eliminating platform dependency.
* **Object-Oriented Programming (OOP):** Java is built on OOP principles, making it modular, reusable, and easy to understand.
* **Secure and Robust:** Java provides features like automatic memory management, exception handling, and runtime checks, making it a reliable choice.
* **Versatile:** Java is used in various domains, including web development, mobile applications (Android), enterprise systems, and IoT devices.

**Why Java Matters**

Java revolutionized programming by emphasizing portability, security, and simplicity. Over the decades, it has remained relevant due to its adaptability, vibrant community, and regular updates. From beginner-friendly applications to complex enterprise systems, Java continues to shape the tech landscape.

**Flavors of Java:**

Java comes in different "flavors" or editions, each designed for specific use cases and applications. These flavors make Java versatile and adaptable, allowing developers to choose the right version for their needs. Let’s explore them:

**1. Java Standard Edition (Java SE)**

* **What is it?** Java SE is the foundation of all Java programming. It provides the core functionalities required to develop general-purpose applications.
* **Key Features:**
  + Basic programming tools (e.g., Java Development Kit - JDK).
  + Core libraries such as collections, multithreading, networking, and database connectivity.
  + APIs for creating desktop applications.
  + Java Virtual Machine (JVM) for running Java programs.
* **Where is it used?**
  + Standalone desktop applications.
  + Basic server-side applications.
  + Learning and teaching Java programming.
* **Examples:**
  + Calculator applications.
  + Text editors.
  + Scientific tools like simulations.

**2. Java Enterprise Edition (Java EE) *(Now Jakarta EE)***

* **What is it?** Java EE extends Java SE by adding libraries and tools specifically designed for building large-scale, distributed, and multi-tier enterprise applications.
* **Key Features:**
  + Support for web services and APIs.
  + Tools for creating server-side applications, including Servlets, JSP (Java Server Pages), and Enterprise JavaBeans (EJB).
  + Scalability and high-performance features for enterprise-level applications.
* **Where is it used?**
  + Large enterprise systems (e.g., banking, e-commerce platforms).
  + Web and cloud-based applications.
  + Backend systems for handling large user bases.
* **Examples:**
  + E-commerce platforms like Amazon.
  + Banking systems.
  + Online ticketing systems.

**3. Java Micro Edition (Java ME)**

* **What is it?** Java ME is a lightweight edition of Java designed for resource-constrained devices such as mobile phones, embedded systems, and IoT (Internet of Things) devices.
* **Key Features:**
  + Optimized for devices with limited memory and processing power.
  + Provides APIs for device-specific functionalities like sensors, GPS, and low-level hardware access.
  + Compact version of Java libraries.
* **Where is it used?**
  + Mobile phones (especially in the pre-smartphone era).
  + Embedded devices like smartcards and set-top boxes.
  + IoT devices like smart thermostats and wearables.
* **Examples:**
  + Games on old mobile phones (e.g., Nokia Snake game).
  + Smart TVs and appliances.
  + IoT solutions like remote sensors.

**4. JavaFX**

* **What is it?** JavaFX is a modern framework for building rich Internet applications (RIAs) with advanced graphical user interfaces (GUIs).
* **Key Features:**
  + Provides a set of graphics and media APIs.
  + Supports CSS styling and FXML for designing user interfaces.
  + Better graphics capabilities compared to Swing and AWT (earlier GUI frameworks).
* **Where is it used?**
  + Applications requiring interactive user interfaces.
  + Data visualization tools.
  + Multimedia applications.
* **Examples:**
  + Interactive dashboards.
  + Media players.
  + Educational apps with animations.

**5. Java Card**

* **What is it?** Java Card is a specialized edition of Java designed for smartcards and other small devices that run secure applications.
* **Key Features:**
  + Extremely lightweight and secure.
  + Supports cryptographic operations like encryption and digital signatures.
  + Runs in highly constrained environments.
* **Where is it used?**
  + Smartcards (e.g., SIM cards, credit cards).
  + Security tokens.
  + Secure identification systems.
* **Examples:**
  + Contactless payment cards.
  + Biometric ID cards.
  + Secure access badges.

**Choosing the Right Flavor**

The choice of Java edition depends on the type of application you are building:

* For general-purpose programs, go with **Java SE**.
* For large-scale enterprise systems, choose **Java EE**.
* For lightweight, resource-constrained devices, **Java ME** is ideal.
* For graphical and interactive applications, use **JavaFX**.
* For secure and portable applications on small devices, opt for **Java Card**.
* **Versions of Java:**

Java has gone through multiple versions since its inception, each bringing new features, enhancements, and tools to make the language more robust and efficient. Let’s explore the key versions of Java and their highlights.

**1. Java 1.0 (1996)**

* **The Beginning of Java**
  + Released by Sun Microsystems.
  + Introduced the concept of "Write Once, Run Anywhere" (WORA).
  + Included core features like applets for creating web-based interactive content.
  + Had basic APIs such as java.lang, java.util, and java.io.

**2. Java 1.1 (1997)**

* **Enhanced Functionality**
  + Added inner classes for better object-oriented programming.
  + Introduced JDBC (Java Database Connectivity) for database access.
  + Improved security and introduced the java.awt package for GUI development.

**3. Java 1.2 (1998) *(Also known as Java 2)***

* **Java 2 Platform**
  + Marked a major overhaul, rebranding as **Java 2**.
  + Introduced the Swing library for more advanced GUIs.
  + Added the Collections Framework for efficient data handling.
  + Improved performance with the Just-In-Time (JIT) compiler.

**4. Java 1.3 (2000)**

* **Stabilization and Improvements**
  + Focused on performance and stability.
  + Added new APIs like JavaSound for audio processing.
  + Enhanced RMI (Remote Method Invocation) and CORBA for distributed computing.

**5. Java 1.4 (2002)**

* **Enterprise Focus**
  + Introduced the **assert** keyword for debugging.
  + Added NIO (New Input/Output) for better file handling and networking.
  + Enhanced XML processing and logging capabilities.

**6. Java 5 (2004) *(Previously Java 1.5)***

* **Significant Language Updates**
  + Introduced generics for type-safe collections.
  + Added features like enhanced for-loops, autoboxing/unboxing, and enumerations.
  + Introduced annotations for metadata.
  + Brought the java.util.concurrent package for better multithreading.

**7. Java 6 (2006)**

* **Performance and Web Enhancements**
  + Improved performance and diagnostics.
  + Added support for scripting languages like JavaScript through the javax.script API.
  + Enhanced web services and database support.

**8. Java 7 (2011)**

* **Improved Syntax and Features**
  + Introduced features like try-with-resources for better exception handling.
  + Added the diamond operator (<>) for cleaner code with generics.
  + Included the NIO.2 framework for enhanced file handling.
  + Improved support for dynamic languages through the invokedynamic instruction.

**9. Java 8 (2014)**

* **A Revolutionary Release**
  + Introduced **Lambda Expressions** for functional programming.
  + Added the **Stream API** for processing collections.
  + Introduced the new **Date and Time API** (java.time).
  + Brought in default methods for interfaces to allow backward compatibility.

**10. Java 9 (2017)**

* **Modularity and Improvements**
  + Introduced the **Java Platform Module System (JPMS)**, also known as Project Jigsaw.
  + Added JShell, an interactive REPL tool for testing code snippets.
  + Improved performance and added APIs like ProcessHandle for managing OS processes.

**11. Java 10 (2018)**

* **Simplified Development**
  + Introduced the var keyword for local variable type inference.
  + Enhanced garbage collection with the G1 Garbage Collector.

**12. Java 11 (2018)**

* **LTS (Long-Term Support) Version**
  + Java 11 became an LTS release, ensuring long-term support.
  + Removed the Java applet and JavaFX from the core distribution.
  + Added new APIs like HttpClient for handling HTTP requests.

**13. Java 12 (2019)**

* **Minor Enhancements**
  + Added switch expressions for cleaner switch statements.
  + Improved performance with JVM optimizations.

**14. Java 13 (2019)**

* **Code Readability**
  + Introduced text blocks for easier multi-line string handling.

**15. Java 14 (2020)**

* **Preview Features**
  + Added features like records (data classes) and pattern matching (for instanceof).
  + Continued improvements to switch expressions.

**16. Java 15 (2020)**

* **Security and Modern Features**
  + Added sealed classes for better control of inheritance.
  + Improved cryptographic algorithms.

**17. Java 16 (2021)**

* **Stability Enhancements**
  + Improved memory management.
  + Added vector API for performance optimization.

**18. Java 17 (2021)**

* **Current LTS Version**
  + Simplified development with improved APIs and features.
  + Enhanced garbage collection and performance.

**Ongoing Releases**

* Oracle shifted Java to a **6-month release cycle** starting with Java 9. While every version brings incremental improvements, only certain versions (like Java 11 and 17) are designated as **LTS (Long-Term Support)** releases, making them more suitable for production environments.

**What is a Compiler?**

A **compiler** is a special program that translates the source code written in a high-level programming language (like Java) into machine code (or bytecode, in the case of Java) that a computer can execute.

In simple terms, the compiler acts as a translator between human-readable code and the code that the computer understands.

**Steps in Compilation:**

1. **Source Code:** You write your Java program in a .java file (plain text).
2. **Lexical Analysis:** The compiler reads the source code and breaks it into smaller parts, called tokens (keywords, operators, etc.).
3. **Syntax Analysis:** It checks the code for grammatical correctness (syntax).
4. **Semantic Analysis:** The compiler ensures that the code has logical correctness (e.g., making sure variables are properly defined).
5. **Intermediate Code Generation:** The compiler translates the source code into an intermediate form (Java bytecode in this case).
6. **Optimization:** The compiler may optimize the bytecode to make it run more efficiently.
7. **Bytecode Generation:** Finally, it creates the **.class file** containing the bytecode

This is what the Java Virtual Machine (JVM) can execute.

**How JVM Works in Java?**

The **Java Virtual Machine (JVM)** is an engine that provides a runtime environment to execute Java bytecode. Unlike a compiler that translates code into machine code for a specific system, the JVM allows Java to be **platform-independent**, meaning you can run the same bytecode on any platform that has a JVM.

Here’s a simple breakdown of how JVM works:

**Steps in JVM Execution:**

1. **Write the Java Code:**
   * You write your Java program in a .java file.
2. **Compile the Java Code:**
   * The Java compiler (javac) takes the .java file and compiles it into **bytecode**, which is stored in a .class file. Bytecode is an intermediate form that is platform-independent.
3. **Load Bytecode into JVM:**
   * The JVM loads the .class file (bytecode) into its memory.
   * It uses a **Class Loader** to load the bytecode into the JVM’s memory space.
4. **JVM Execution:**
   * The **JVM's Execution Engine** reads and executes the bytecode.
   * There are two main approaches:
     + **Interpreter:** The JVM reads the bytecode line by line and converts it to machine code.
     + **Just-In-Time (JIT) Compiler:** The JVM compiles the bytecode into native machine code at runtime (while the program is running). This improves performance as parts of the code that are frequently executed get compiled into machine code for faster execution.
5. **Garbage Collection:**
   * The JVM automatically handles memory management. It keeps track of objects that are no longer in use and frees up memory, a process called **garbage collection**.
6. **Output:**
   * The JVM executes the bytecode, and the program produces output as expected, whether it’s data processing, interacting with a user, or accessing a database.

**Why is JVM Important?**

* **Platform Independence:** Java’s core principle is “Write Once, Run Anywhere” (WORA). The JVM allows this by enabling the same bytecode to run on any platform with a JVM installed (Windows, macOS, Linux, etc.).
* **Security:** The JVM helps manage memory, performs bytecode verification, and uses sandboxes to prevent unauthorized access to system resources.
* **Optimization:** The JVM can optimize performance using techniques like Just-In-Time (JIT) compilation, making Java programs run efficiently.

**JVM Components:**

1. **Class Loader:**
   * Loads Java classes into memory.
2. **Runtime Data Areas:**
   * **Method Area:** Stores class-level data, like method information and static variables.
   * **Heap:** Stores objects and arrays created during runtime.
   * **Stack:** Stores method calls, local variables, and partial results.
   * **Program Counter Register:** Keeps track of the current instruction being executed.
   * **Native Method Stack:** Used for native (non-Java) code execution.
3. **Execution Engine:**
   * It interprets bytecode or uses the JIT compiler to convert bytecode into machine code.
   * Handles the actual execution of the bytecode instructions.
4. **Garbage Collector:**
   * Automatically reclaims memory by removing objects that are no longer needed.

**Summary of How Java Code Runs:**

1. You write Java code in a .java file.
2. The Java compiler (javac) converts the source code into bytecode, which is stored in a .class file.
3. The JVM loads the bytecode and executes it, either interpreting it line by line or using a JIT compiler to convert it to native machine code.
4. The JVM manages memory and handles garbage collection to optimize the execution.

**Write Your First Java Program**

public class HelloWorld {

public static void main(String[] args) {

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

}

}

**Explanation of the Code:**

* public class HelloWorld: This is the class declaration. Every Java program must have at least one class.
* public static void main(String[] args): This is the **main method**, where execution starts. It's like the entry point of the program.
* System.out.println("Hello, World!");: This line prints **"Hello, World!"** to the console.

**Compile the Program**:

* Type **javac** **HelloWorld.java** and press **Enter**.
* This generates the **HelloWorld.class** bytecode file if there are no syntax errors.

**Run the Program**:

* Type **java HelloWorld** and press **Enter**.
* The output should be:

Hello, World!

* **What is a Class in Java?**

A **class** in Java is a blueprint or template used to create objects. It defines the structure and behavior that objects of that class will have.

**Key Points:**

* **Definition**: A class contains variables (called **fields**) to store data and methods (called **behaviors**) to define actions that objects can perform.
* **Structure**:
  + **Fields**: Variables that hold data specific to an object.
  + **Methods**: Functions that define actions or behaviors for the object.

**Creating an Object:**

* An object is an instance of a class. When you create an object, you allocate memory for its fields and can call its methods.

**Example:**

public class Car {

// Fields

String color;

String model;

// Method

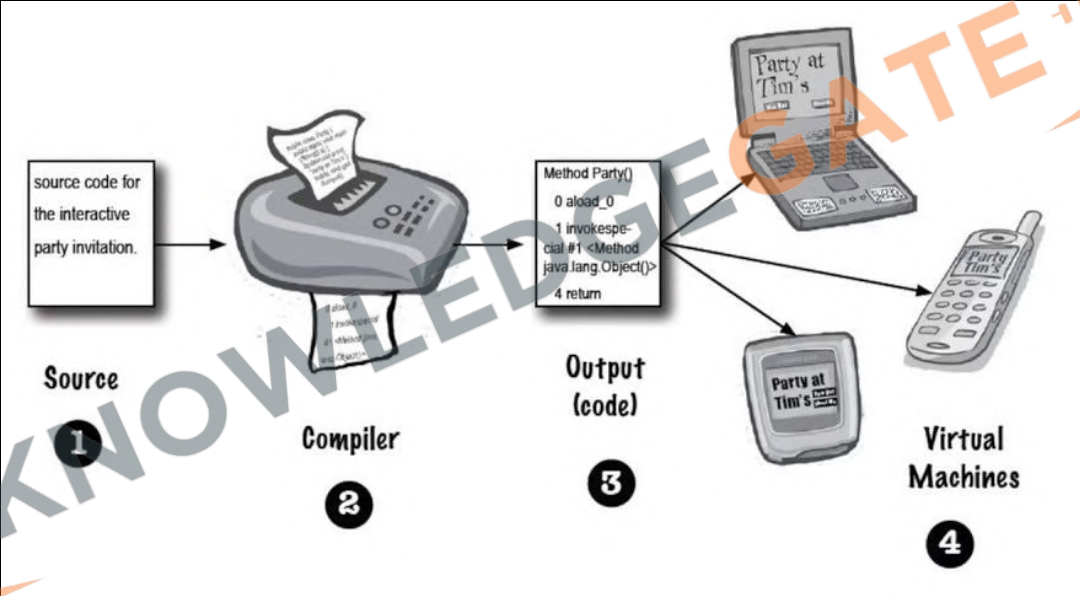
void startEngine() {

System.out.println("Engine started");

}

}

* **Car** is the class.
* **color** and **model** are fields.
* **startEngine()** is a method.
* **Magic of Byte Code**



**Magic of Bytecode in Java: Diagram Explanation**

1. **Source Code (Step 1)**
   * Java program source code is written in a .java file.
   * Example: A program to create an interactive invitation.
2. **Compiler (Step 2)**
   * The Java Compiler (javac) compiles the source code into **bytecode**.
   * Bytecode is stored in a .class file.
3. **Output (Code) (Step 3)**
   * The bytecode consists of machine-independent instructions.
   * Example: Instructions like aload, invoke, and return.
4. **Virtual Machines (Step 4)**
   * The **JVM (Java Virtual Machine)** reads and executes bytecode.
   * JVM is available for different devices, like:
     + **Laptops .Mobile phones , Handheld devices**
   * This makes Java **platform-independent**:  
     *"Write Once, Run Anywhere."*

**Key Points**

* **Bytecode** is the intermediate language between source code and machine code.
* The **JVM** translates bytecode into machine code specific to each device.
* This enables Java programs to run on any device with a JVM installed
* **How Java Changed the Internet**

Java changed the internet by enabling **dynamic and interactive web content**. Unlike static HTML pages, Java introduced applets—small programs that could run directly in a web browser. This allowed developers to create features like **games, animations, and real-time updates** on websites.

Key contributions:

1. **Platform independence:** Java's "Write Once, Run Anywhere" capability worked across all operating systems.
2. **Security:** Java provided a secure environment for running untrusted code (sandbox).
3. **Networking:** Java simplified building internet-based applications with its built-in networking libraries.

* **Java Buzzwords**

Java buzzwords are the core features that make Java unique and powerful. These terms highlight Java's design principles and functionality. Here's a simple explanation of each buzzword:

**1. Simple**

* **Meaning**: Java is easy to learn and use.
* **Why?**: It has a clean syntax and removes complex features like pointers (used in C++).

**2. Object-Oriented**

* **Meaning**: Everything in Java revolves around objects.
* **Why?**: It uses concepts like classes, objects, inheritance, encapsulation, and polymorphism.

**3. Platform-Independent**

* **Meaning**: Write once, run anywhere (WORA).
* **Why?**: Java programs compile into bytecode, which runs on any device with a Java Virtual Machine (JVM).

**4. Secure**

* **Meaning**: Java provides built-in security features.
* **Why?**: It has no explicit pointers and runs code inside a secure JVM environment, preventing unauthorized access.

**5. Robust**

* **Meaning**: Java handles errors efficiently.
* **Why?**: It has strong memory management, exception handling, and automatic garbage collection

**6. Architecture-Neutral**

* **Meaning**: Java code works on any architecture (e.g., 32-bit or 64-bit systems).
* **Why?**: Bytecode is designed to be compatible across different platforms.

**7. Portable**

* **Meaning**: Java programs are easily moved from one system to another.
* **Why?**: Java eliminates platform-specific dependencies.

**8. High Performance**

* **Meaning**: Java performs well, though not as fast as C++.
* **Why?**: The Just-In-Time (JIT) compiler optimizes code execution.

**9. Multithreaded**

* **Meaning**: Java can run multiple threads simultaneously.
* **Why?**: It allows better CPU utilization and handles tasks like animation or background operations.

**10. Distributed**

* **Meaning**: Java supports distributed systems and networking.
* **Why?**: It has APIs like RMI (Remote Method Invocation) and supports sharing data over networks.

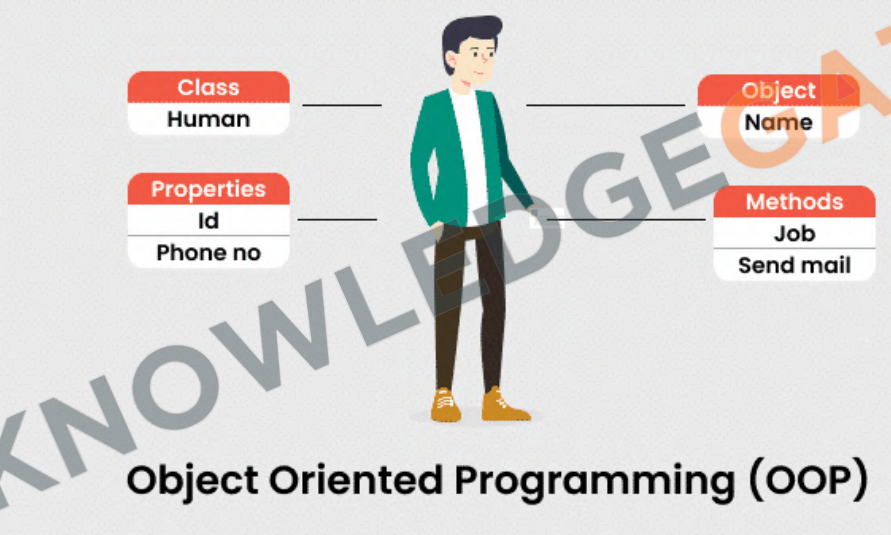
**11. Dynamic**

* **Meaning**: Java can adapt to changing environments.
* **Why?**: It loads classes at runtime and links them dynamically.
* **Object-Oriented Programming (OOP)**

**Object-Oriented Programming (OOP)** is a programming paradigm that organizes code into **objects**, which are instances of **classes**. Here are the important points:

1. **Classes and Objects**:
   * **Class**: A blueprint for creating objects, defining properties (attributes) and methods (functions).
   * **Object**: An instance of a class that holds specific data and can perform actions defined by its class.
2. **Encapsulation**:
   * Bundling data (variables) and methods (functions) that operate on the data within a single unit (class).
   * Restricts direct access to some of the object’s components, using access modifiers (private, public, protected).
3. **Abstraction**:
   * Hiding the complex implementation details and showing only the necessary features.
   * Achieved through abstract classes and interfaces.
4. **Inheritance**:
   * A mechanism where one class (child) inherits properties and behaviors from another class (parent).
   * Promotes code reusability and hierarchical relationships between classes.
5. **Polymorphism**:
   * The ability of different objects to respond to the same method in a way appropriate to their type.
   * Achieved through **method overloading** (same method name, different parameters) and **method overriding** (same method signature, different implementation).

OOP helps in improving code reusability, maintainability, and scalability. It is widely used in many programming languages like Java, C++, Python, and C#.



* **JAVA BASICS**

First Program – Hello World

Ex –

public class Hello {

public static void main(String[] args) {

System.out.println("Hello World My Name Is Amol Kadam ?...");

}

}

To run this code:

1. Save it in a file named **Hello.java**
2. Compile it using **javac Hello.java**
3. Run the program using **java Hello**

This is a simple Java program that prints a message to the console.

**Explanation:**

1. **class Hello**:  
   This defines a class named Hello. In Java, a program is structured in classes, and the class contains methods to execute code.
2. **public static void main(String[] args)**:
   * **public**: This means the main method can be accessed from outside the class.
   * **static**: This means that the method belongs to the class itself, not to an instance of the class. This allows the program to run without creating an object of the class.
   * **void**: This means the main method doesn't return any value.
   * **main(String[] args)**: This is the entry point of the Java program. When you run the program, the code inside this method is executed. args can be used to accept command-line arguments.
3. **System.out.println("Hello World My Name Is Amol Kadam ?...");**:  
   This line prints the message "Hello World My Name Is Amol Kadam ?..." to the console.
   * **System**: Refers to the built-in class in Java.
   * **out**: Refers to the standard output stream (console).
   * **println**: This method prints the text and moves the cursor to a new line after printing.

**Summary:**

When you run this program, it will print:

O/P :

Hello World My Name Is Amol Kadam ?...

**1. Compiling Java Code:**

* The Java code needs to be translated from human-readable text into bytecode, which the Java Virtual Machine (JVM) can execute.
* **Command:**

**javac <filename>.java**

* + This command uses the javac compiler to convert the source code file (with .java extension) into a bytecode file (with .class extension). The .class file contains the bytecode that the JVM can execute.
* Example:

**javac HelloWorld.java**

* This will generate a **HelloWorld.class** file, which contains the compiled bytecode.

**2. Running the Java Program:**

* After compiling the program, you need to run the bytecode using the JVM.
* **Command:**

**java <classname>**

* + This command runs the compiled bytecode (the .class file) on the JVM. Make sure not to include the .class extension when running the program.
* Example:

**java HelloWorld**

* The output of the program will be displayed in the terminal or command prompt.

**Example Workflow:**

1. Write the Java program:

public class HelloWorld {

public static void main(String[] args) {

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

}

}

1. Save the file as HelloWorld.java.
2. Open the terminal and navigate to the folder where the file is saved.
3. Compile the program:

**javac HelloWorld.java**

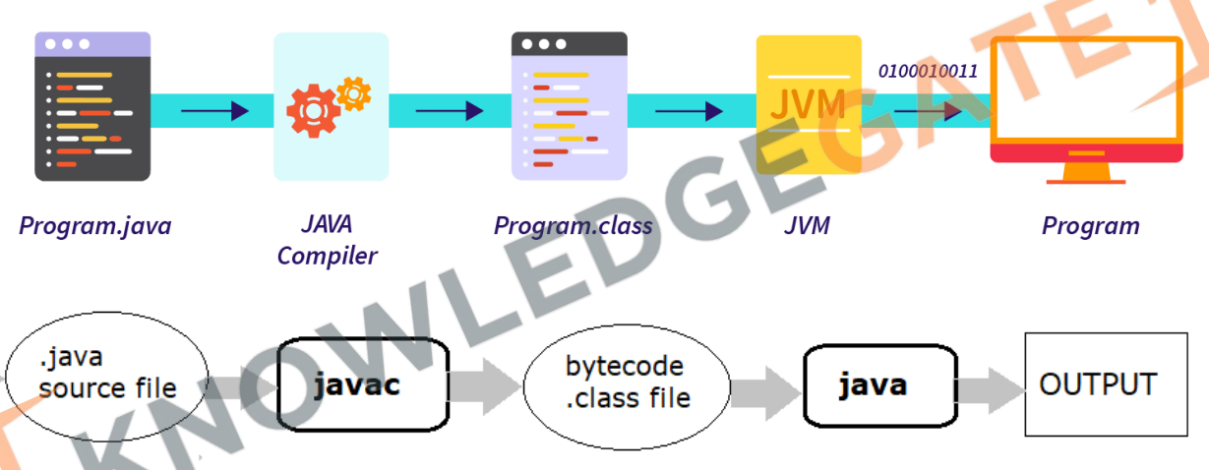
1. Run the compiled program:

**java HelloWorld**

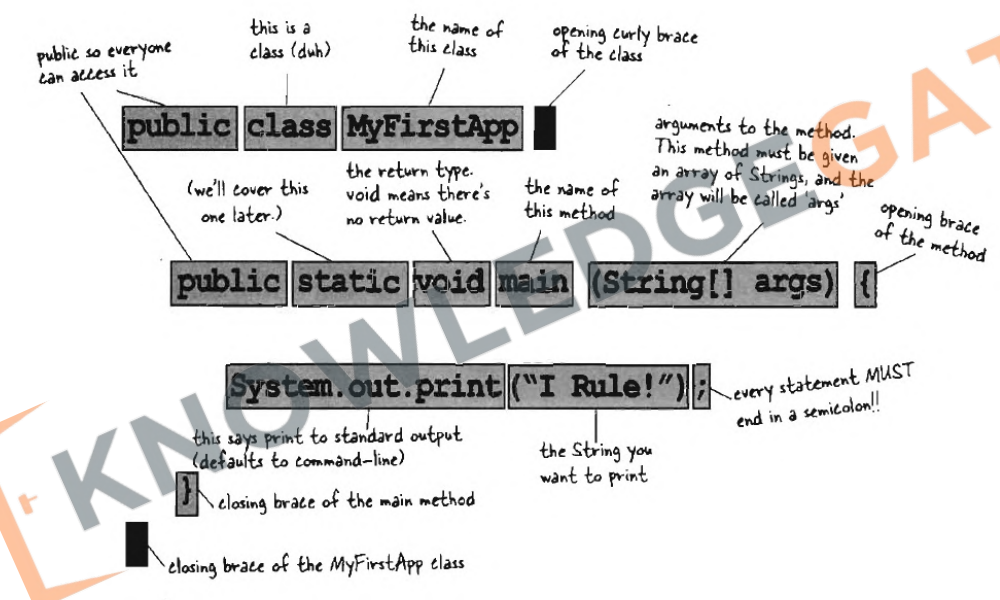
**Output:**

**Hello, World!**

This is a basic process for compiling and running Java programs.



* **Anatomy of a Class**

****

* **File Extensions**

**.Java File:**

* **Contains Java source code written in a human-readable format.**
* **Used for development and programming.**
* **Can be edited using text editors or IDEs.**
* **Needs to be compiled into bytecode (.class) before execution.**

**.Class File:**

* **Contains Java bytecode (compiled form of .java file).**
* **Meant for JVM (Java Virtual Machine) to execute the program.**
* **Not editable, as it is the output of the compilation process.**
* **Used for execution of the Java application.**
* **JDK vs JVM vs JRE**

**JDK (Java Development Kit):**

* What it is: A software package required to develop Java applications.
* Includes: Everything you need for coding, compiling, debugging, and running Java applications (includes JRE, compiler javac, interpreter java, and other tools).
* For Developers: Essential for anyone writing Java code.
* Example: If you're developing a Java program, you need the JDK to both write and run it.

**JRE (Java Runtime Environment):**

* What it is: A part of the JDK, but can be downloaded separately. It provides the necessary components to run Java applications.
* Includes: JVM, Java libraries, and other components to execute Java bytecode.
* For Users: Needed for running Java applications but not for development.
* Example: If you only need to run a Java program, you only need the JRE.

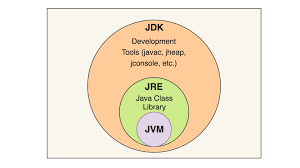
**JVM (Java Virtual Machine):**

* What it is: The engine that runs Java bytecode on your system.
* Function: Ensures Java’s "write once, run anywhere" capability by converting bytecode into machine-specific code and executing it.
* Platform-Specific: There’s a different JVM for each operating system (Windows, macOS, Linux).
* Example: If you write a Java program on Windows, the JVM ensures it can run on any other OS as long as the corresponding JVM is available.

**Summary:**

* **JDK** = All-in-one package for developing Java apps (includes JRE + development tools).
* **JRE** = Environment to run Java apps (includes JVM + libraries, but no development tools).
* **JVM** = The core engine that runs Java programs on any device.

If you’re a developer, you’ll use the JDK. If you just need to run Java programs, you’ll need the JRE.



* **Showing Output**
* **System.out.println("message");**
  + **This prints the message and then moves to the next line.**
  + **The println() method stands for "print line", so it prints the message followed by a line break.**
* **System.out.print("message");**
  + **This prints the message but does not move to the next line after printing.**
  + **The print() method simply prints the message without any additional line break.**

**Example:**

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

**System.out.print("World");**

**System.out.println("!");**

**Output:**

**Hello**

**World!**

* **println adds a line break after "Hello".**
* **print keeps "World" on the same line.**
* **println adds a line break after "!".**

**So, println is used when you want each message on a new line, and print keeps the output on the same line.**

* **Importance of the Main Method**

1. **Entry Point:**
   * **The main method is the starting point of a Java program.**
   * **When you run a Java program, the JVM (Java Virtual Machine) looks for the main method to begin execution. Without it, the JVM wouldn't know where to start.**
2. **Public and Static:**
   * **Public: The main method is declared as public, meaning it can be accessed from anywhere, including the JVM which needs to call it to run the program.**
   * **Static: It's static so that it can be called without creating an instance of the class. The JVM doesn't create an object to invoke the main method.**
3. **Fixed Signature:**
   * **The main method has a specific structure or signature: public static void main(String[] args).**
   * **The method must exactly match this signature, as the JVM recognizes this as the valid format for starting the program.**
   * **If you change the method's signature, such as using different parameters or changing void, the JVM won’t be able to find the starting point.**

**This method is essential because it acts as the "entry door" for running your Java application.**

* **IDE (Integrated Development Environment)**

**IDE (Integrated Development Environment)** is a software application that combines all the tools needed for software development into a single platform. It helps developers write, test, and debug code more efficiently.

**Need of IDE**

1. **Streamlines Development**: It combines all necessary tools, so you don't need to switch between different applications.
2. **Increases Productivity**: By automating tasks like error detection, it saves time.
3. **Simplifies Complex Tasks**: Makes tasks like debugging or testing easier with built-in features.
4. **Offers a Unified Workspace**: Everything is in one place, making it easier to manage projects.

**IDE Features**

1. **Code Autocomplete**: Suggests code completions as you type, speeding up coding.
2. **Syntax Highlighting**: Colors different parts of the code to make it easier to read and spot errors.
3. **Version Control**: Tracks code changes, helping manage different versions of a project.
4. **Error Checking**: Identifies errors in real-time, allowing you to fix them early.

In short, an IDE is designed to make coding faster, easier, and less error-prone.

Imagine you're writing a Java program in an IDE like **IntelliJ IDEA** or **Eclipse**

* **What are Variables?**
* **Definition:** Variables are like containers in programming that store data values in a computer's memory.
* **Purpose:** They allow you to label and store data for use in your program.

**Key Points About Variables:**

1. **Name:**
   * Each variable has a unique name (identifier) used to access the stored value.
   * Example: int age = 25;  
     Here, age is the variable name.
2. **Type:**
   * Variables have a specific data type, such as int, float, char, etc., that defines what kind of value they can store.
   * Example:
     + int → for integers like 5, 10
     + double → for decimal numbers like 3.14
     + char → for single characters like 'A'.
3. **Memory Allocation:**
   * When a variable is declared, a specific part of the memory is reserved to store its value.
4. **Initialization:**
   * A variable must be assigned a value before it is used.
   * Example: int age = 25; (declared and initialized in one step).

**How to Declare a Variable**

**Syntax:**

data\_type variable\_name = value;

**Example:**

int number = 10;

// 'int' is the data type, 'number' is the variable name, and '10' is the value.

**Types of Variables in Java**

1. **Local Variables:**
   * Declared inside a method or block.
   * Only accessible within that block.
   * **Example:**

void myMethod() {

int localVar = 5; // Local variable

}

1. **Instance Variables:**
   * Declared inside a class but outside any method.
   * Belongs to an instance of the class.
   * **Example:**

class Example {

int instanceVar = 10; // Instance variable

}

1. **Static Variables:**
   * Declared with the static keyword.
   * Shared among all instances of the class.
   * **Example:**

class Example {

static int staticVar = 20; // Static variable

}

**Rules for Naming Variables (Identifiers):**

1. Must start with a letter (A-Z, a-z), $, or \_.
2. Cannot use Java reserved keywords (e.g., int, class).
3. Case-sensitive (e.g., Age and age are different).

**Example Program:**

public class Main {

public static void main(String[] args) {

int age = 25; // Variable to store age

double salary = 50000; // Variable to store salary

char grade = 'A'; // Variable to store grade

System.out.println("Age: " + age);

System.out.println("Salary: " + salary);

System.out.println("Grade: " + grade);

} }

**Quick Summary**

* **Variable = Data container** (stores data in memory).
* **Declaration Syntax:** datatype variable-name = value;
* **Types:** Local, Instance, Static.
* **Rules:** Name must start with a letter, \_, or $; avoid keywords; case-sensitive.
* **Java Data Types**

Java has two main types of data types:

1. **Primitive Data Types**
2. **Non-Primitive Data Types**

**1. Primitive Data Types**

These are the most basic data types in Java, predefined by the language. There are **8 types** of primitive data types:

1. **byte**
   * **Size**: 1 byte (8 bits)
   * **Range**: -128 to 127
   * **Example**: byte age = 25;
   * **Use**: Efficient memory usage for small integer values.
2. **short**
   * **Size**: 2 bytes (16 bits)
   * **Range**: -32,768 to 32,767
   * **Example**: short year = 2023;
   * **Use**: Larger integer values than byte but smaller than int.
3. **int**
   * **Size**: 4 bytes (32 bits)
   * **Range**: -2,147,483,648 to 2,147,483,647
   * **Example**: int salary = 50000;
   * **Use**: Default choice for integer values in Java.
4. **long**
   * **Size**: 8 bytes (64 bits)
   * **Range**: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
   * **Example**: long distance = 9876543210L;
   * **Use**: Very large integer values (append L to the value).
5. **float**
   * **Size**: 4 bytes (32 bits)
   * **Range**: ~3.4e-038 to 3.4e+038 (7 decimal precision)
   * **Example**: float price = 19.99F;
   * **Use**: Decimal numbers with less precision (append F).
6. **double**
   * **Size**: 8 bytes (64 bits)
   * **Range**: ~1.7e-308 to 1.7e+308 (15 decimal precision)
   * **Example**: double pi = 3.14159265359;
   * **Use**: High-precision decimal values.
7. **char**
   * **Size**: 2 bytes (16 bits)
   * **Range**: Unicode characters ('\u0000' to '\uffff')
   * **Example**: char grade = 'A';
   * **Use**: Single characters or Unicode values.
8. **boolean**
   * **Size**: 1 bit (true/false)
   * **Values**: true or false
   * **Example**: boolean isJavaFun = true;
   * **Use**: Logical conditions and decision-making.

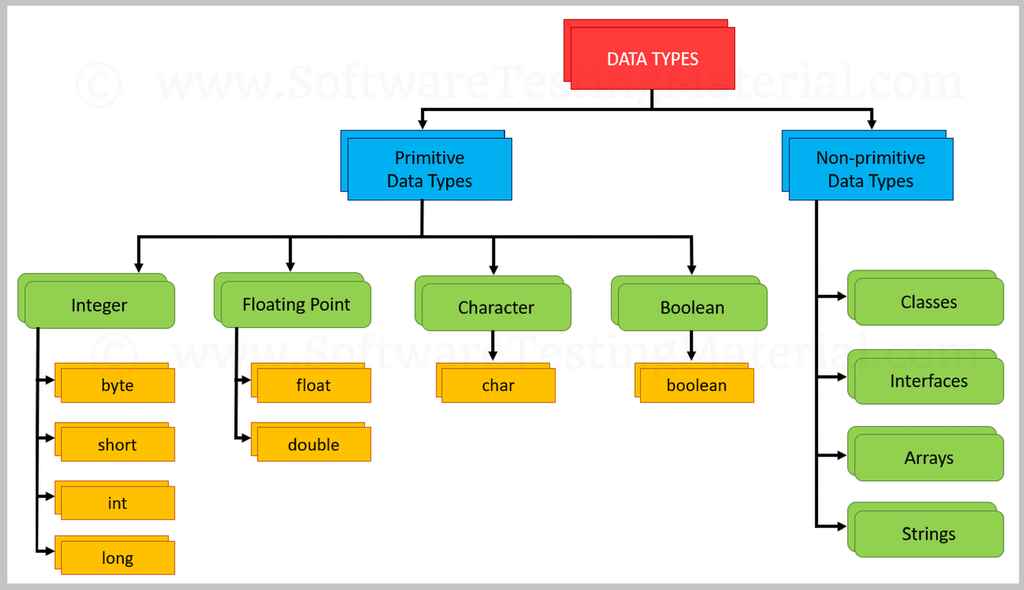
**2. Non-Primitive Data Types**

These are more complex data types created by the programmer. Examples include **Strings**, **Arrays**, **Classes**, and **Interfaces**.

1. **String**
   * A sequence of characters.
   * Example: String name = "Java";
   * Use: Represent text data.
2. **Array**
   * A collection of elements of the same data type.
   * Example: int[] numbers = {1, 2, 3, 4};
   * Use: Store multiple values in a single variable.
3. **Class**
   * A blueprint for objects.
   * Example: class Car { String color; int speed; }
4. **Interface**
   * A blueprint of a class containing only abstract methods.

**Key Points to Remember**

* **Default Values**:
  + byte, short, int, long → 0
  + float, double → 0.0
  + char → '\u0000'
  + boolean → false
  + Non-primitive types → null
* **Type Casting**:
  + Implicit (smaller to larger): int num = 10; long bigNum = num;
  + Explicit (larger to smaller): double value = 9.7; int num = (int) value;
* Use appropriate data types based on memory needs and value ranges.



* **Naming Conventions in Java**

Naming conventions in Java are essential to writing clean, readable, and maintainable code. Below are the key styles used for naming variables, methods, and other elements in Java:

**1. camelCase**

* **Definition**: In camelCase, the name starts with a lowercase letter, and the first letter of each subsequent word is capitalized.
* **Where It's Used**: Commonly used for variable names, method names, and object names.
* **How to Use**:
  + Start with a lowercase letter.
  + Capitalize the first letter of every word after the first.
  + Avoid using spaces or underscores.
* **Example**:

int myVariableName = 5; // Good naming

int myvariable = 5; // Bad naming (hard to read)

* **Key Tip**: camelCase helps in keeping variable and method names concise while improving readability.

**2. snake\_case**

* **Definition**: In snake\_case, all letters are lowercase, and words are separated by underscores (\_).
* **Where It's Used**: Rarely used in Java, but sometimes seen in database schemas or constants in specific styles.
* **How to Use**:
  + Use only lowercase letters.
  + Separate each word with an underscore (\_).
* **Example**:

int my\_variable\_name = 10; // Rare in Java, more common in other languages like Python.

**3. kebab-case**

* **Definition**: In kebab-case, all letters are lowercase, and words are separated by hyphens (-).
* **Where It's Used**: Not used in Java for variable names, as hyphens are not allowed in identifiers. However, it's commonly used in file names or URLs.
* **How to Use**:
  + Use only lowercase letters.
  + Separate words with hyphens (-).
* **Example**:  
  **my-variable-name** (Not valid in Java code, but might appear in file paths or configuration files).

**4. Keep Names Short and Descriptive**

* Choose names that are clear and describe the purpose of the variable, method, or class without being overly long.
* **Why?**: Short names save time while coding, and descriptive names make the code easy to understand.
* **Good Practices**:
  + Use meaningful words (e.g., count, totalPrice, userAge).
  + Avoid abbreviations unless they are universally understood (e.g., int temp is okay for temperature).
* **Example**:

// Good naming:

int totalStudents = 50; // Descriptive and meaningful.

// Bad naming:

int ts = 50; // Not clear, what does "ts" mean?

**Important Points to Remember**

1. **Java is Case-Sensitive**:
   * **myvariable** and **myVariable** are two different identifiers in Java.
2. **Avoid Reserved Words**:
   * You cannot use Java keywords like **int**, **class**, **if**, or while as variable names.
3. **Follow the Context**:
   * Variables: Use camelCase (e.g., totalPrice).
   * Constants: Use ALL\_CAPS with underscores (e.g., MAX\_LIMIT).
   * Classes: Use PascalCase (e.g., StudentRecord).
4. **Consistency is Key**:
   * Always stick to one naming convention in your project for uniformity and better teamwork collaboration.

* **Java Identifier Rules -**

Java identifiers are the names used to identify variables, methods, classes, or other elements in a program. To ensure these names are valid, Java has specific rules for writing identifiers. Let’s understand these rules in detail:

**1. Allowed Characters**

* Identifiers can only include **letters (A-Z, a-z)**, **digits (0-9)**, the **dollar sign ($)**, and the **underscore (\_) symbol**.
* **Examples**:  
  ✅ name  
  ✅ price2  
  ✅ \_total  
  ✅ $amount
* Any other symbols like @, %, &, # are **not allowed**.  
  ❌ price@ (invalid)  
  ❌ #value (invalid)

**2. No Keywords or Reserved Words**

* Keywords and reserved words in Java (like class, int, for, etc.) **cannot be used as identifiers** because they have a predefined meaning in the Java language.
* **Examples**:  
  ❌ class (invalid)  
  ✅ myClass (valid)
* **Tip**: If you try to use a keyword, your program will throw a compilation error.

**3. Cannot Start with Digits**

* Identifiers **cannot begin with a digit** (0-9). However, digits can be used after the first character.
* **Examples**:  
  ✅ value1 (valid)  
  ❌ 1value (invalid)

**4. Case Sensitivity**

* Java identifiers are **case-sensitive**, meaning uppercase and lowercase letters are treated as different characters.
* **Examples**:  
  total, Total, and TOTAL are **three different identifiers** in Java.

**5. Length of Identifiers**

* There is **no limit** to how long an identifier can be. However, it’s best to use a length between **4 to 15 characters** for readability and convenience.
* **Examples**:  
  ✅ userID (good length)  
  ✅ totalMarks (good length)  
  ❌ totalMarksObtainedByTheStudentInSubject (too long)
* **Summary Table of Rules**

|  |  |  |
| --- | --- | --- |
| Rule | Example (Valid) | Example (Invalid) |
| Allowed Characters | age, $value | price# |
| No Keywords | myClass | class |
| No Starting with Digits | value1 | 1value |
| Case Sensitivity | total vs Total | N/A |
| Optimum Length | userID | Too long names |

**Best Practices**

1. Use meaningful names that indicate the purpose of the identifier.  
   Example: Use age instead of a.
2. Follow naming conventions:
   * For variables and methods: **camelCase** (e.g., studentName).
   * For classes: **PascalCase** (e.g., StudentDetails).
3. Avoid starting names with $ or \_ unless necessary.

* **Literals in Java**

Literals in Java are the fixed values or constant values assigned to variables. These values do not change during the program execution. They represent the data you directly use in your code. For example: 5, 'A', "Hello" are all literals.

**Types of Literals in Java:**

1. **Integer Literals**  
   Whole numbers:
   * Decimal: 100
   * Binary: 0b1010 (10 in decimal)
   * Octal: 012 (10 in decimal)
   * Hexadecimal: 0xA (10 in decimal)  
     Default type: int
2. **Floating-point Literals**  
   Decimal numbers:
   * Example: 3.14
   * Scientific: 1.2e3 (1200) , Default type: double , Suffix: f for float (3.14f)
3. **Character Literals**  
   Single characters:
   * Example: 'A', '7' , Special characters: '\\', '\n'
4. **String Literals**  
   Sequence of characters:
   * Example: "Hello World"
5. **Boolean Literals**  
   Represents true or false:
   * Example: boolean isJavaFun = true;
6. **Null Literal**  
   Represents no value:
   * Example: String str = null;
7. **Long Literals**  
   Use L for long:
   * Example: 100L
8. **Underscore in Numeric Literals**  
   For readability:
   * Example: int million = 1\_000\_000;

**Important Notes:**

* Integer literals are int by default.
* Floating-point literals are double by default.
* Use suffix L for long and F for float to specify types.
* If a literal exceeds the range, explicitly cast it.
* **Java keywords**

1. **abstract**: Used to declare an abstract class or method that cannot be instantiated or must be implemented by subclasses.
2. **assert**: Used for debugging, allows you to test assumptions with a condition.
3. **boolean**: Defines a variable that can hold true or false.
4. **break**: Exits a loop or switch statement immediately.
5. **byte**: Defines a variable that can store an 8-bit integer.
6. **case**: Defines a branch in a switch statement.
7. **catch**: Catches and handles exceptions thrown in a try block.
8. **char**: Defines a variable for storing a single 16-bit Unicode character.
9. **class**: Used to declare a class in Java.
10. **const**: Reserved but not used. Historically used for constants.
11. **continue**: Skips the current iteration of a loop and proceeds to the next iteration.
12. **default**: Specifies the default case in a switch statement if no other case matches.
13. **do**: Starts a do-while loop, which runs at least once before checking the condition.
14. **double**: Defines a variable for storing double-precision floating-point numbers.
15. **else**: Defines the alternative block of code in an if-else condition.
16. **enum**: Defines a set of named constants, known as an enum type.
17. **extends**: Indicates that a class is inheriting from another class.
18. **final**: Used to define constants, prevent method overriding, and prevent class inheritance.
19. **finally**: A block of code that executes after a try-catch block, regardless of whether an exception occurs.
20. **float**: Defines a variable for storing single-precision floating-point numbers.
21. **for**: Defines a for loop for iteration.
22. **goto**: Reserved but not used in Java.
23. **if**: Defines a condition for executing code based on boolean expressions.
24. **implements**: Indicates that a class is implementing an interface.
25. **import**: Imports external classes or entire packages to use in the current file.
26. **instanceof**: Tests whether an object is an instance of a specified class or interface.
27. **int**: Defines a variable for storing a 32-bit integer.
28. **interface**: Defines an interface, which can be implemented by classes.
29. **long**: Defines a variable for storing a 64-bit integer.
30. **native**: Specifies that a method is implemented in a language like C or C++.
31. **new**: Creates a new instance of a class (an object).
32. **null**: Represents the absence of a value or an object.
33. **package**: Defines a package (namespace) to group related classes.
34. **private**: Restricts access to a member (variable, method) to the same class only.
35. **protected**: Allows access to a member within the same package or subclasses.
36. **public**: Allows access to a member from any class.
37. **return**: Exits from a method and optionally returns a value.
38. **short**: Defines a variable for storing a 16-bit integer.
39. **static**: Defines a member that belongs to the class rather than instances of the class.
40. **strictfp**: Ensures floating-point calculations follow a strict standard for portability.
41. **super**: Refers to the parent class's methods or constructor.
42. **switch**: Defines a switch statement to check multiple possible conditions.
43. **synchronized**: Ensures that a method or block of code is accessed by only one thread at a time.
44. **this**: Refers to the current object in a method or constructor.
45. **throw**: Used to explicitly throw an exception.
46. **throws**: Declares that a method can throw an exception.
47. **transient**: Prevents serialization of a variable.
48. **try**: Defines a block of code where exceptions are checked and handled.
49. **void**: Defines a method that does not return a value.
50. **volatile**: Indicates that a variable can be modified by multiple threads.
51. **while**: Defines a while loop that continues as long as the condition is true.
52. **yield**: Introduced in Java 14, returns a value in a switch expression.

**Additional keywords introduced in later versions:**

1. **module**: Introduced in Java 9 for defining a module system.
2. **open**: Introduced in Java 9, used for defining open modules that allow deep reflection.
3. **requires**: Introduced in Java 9, specifies module dependencies.
4. **exports**: Introduced in Java 9, defines which packages are available to other modules.
5. **to**: Introduced in Java 9, used for declaring module dependencies.
6. **uses**: Introduced in Java 9, used for declaring a service provider interface.
7. **provides**: Introduced in Java 9, used in the service provider interface declaration.
8. **with**: Introduced in Java 9, used in service provider interface declarations.
9. **transitive**: Introduced in Java 9, used for transitive module dependencies.
10. **var**: Introduced in Java 10, for local variable type inference (i.e., the compiler infers the type).
11. **record**: Introduced in Java 14, used to define immutable data classes.
12. **sealed**: Introduced in Java 15, restricts which classes or interfaces can extend or implement a class.
13. **non-sealed**: Introduced in Java 15, used to extend a sealed class.
14. **permits**: Introduced in Java 15, used with sealed classes to specify permitted subclasses.

These 68 keywords are reserved in Java, meaning they can't be used as identifiers for variables, methods, or classes. They form the backbone of Java syntax, enabling the definition of variables, control structures, classes, and error-handling mechanisms.

* **round and print decimal numbers**

**1. Using Math.round()**

Rounds a floating-point number to the nearest integer.

double num = 3.14159;

System.out.println(Math.round(num)); // Output: 3

**2. Using DecimalFormat**

Allows you to define a specific format for the number.

import java.text.DecimalFormat;

double num = 3.14159;

// Example 1: Round to 2 decimal places

DecimalFormat df = new DecimalFormat("#.##");

System.out.println(df.format(num)); // Output: 3.14

// Example 2: Always show 2 decimal places

DecimalFormat dfFixed = new DecimalFormat("0.00");

System.out.println(dfFixed.format(num)); // Output: 3.14

**3. Using String.format()**

Rounds and formats the number as a string.

double num = 3.14159;

// Example: Round to 2 decimal places

System.out.println(String.format("%.2f", num)); // Output: 3.14

**4. Using BigDecimal**

Provides precise control over rounding.

import java.math.BigDecimal;

import java.math.RoundingMode;

double num = 3.14159;

// Example: Round to 2 decimal places

BigDecimal bd = new BigDecimal(num);

bd = bd.setScale(2, RoundingMode.HALF\_UP);

System.out.println(bd); // Output: 3.14

**5. Using printf**

Formats and directly prints the number.

double num = 3.14159;

// Example: Round to 2 decimal places

System.out.printf("%.2f%n", num); // Output: 3.14

**6. Using Round with Multiplication and Division**

Manually round the number using math.

double num = 3.14159;

// Example: Round to 2 decimal places

double rounded = Math.round(num \* 100.0) / 100.0;

System.out.println(rounded); // Output: 3.14

**7. Using Math.floor() or Math.ceil()**

To always round down or up to a specific decimal place.

double num = 3.14159;

// Example: Round down to 2 decimal places

double floorValue = Math.floor(num \* 100.0) / 100.0;

System.out.println(floorValue); // Output: 3.14

// Example: Round up to 2 decimal places

double ceilValue = Math.ceil(num \* 100.0) / 100.0;

System.out.println(ceilValue); // Output: 3.15

**8. Using NumberFormat**

Handles localization and rounding.

import java.text.NumberFormat;

double num = 3.14159;

// Example: Round to 2 decimal places

NumberFormat nf = NumberFormat.getNumberInstance();

nf.setMaximumFractionDigits(2);

System.out.println(nf.format(num)); // Output: 3.14

**Comparison of Methods**

| **Method** | **Best Use Case** |
| --- | --- |
| Math.round | Quick rounding to the nearest integer. |
| DecimalFormat | Flexible formatting of decimal places. |
| String.format | Formatting strings with decimal precision. |
| BigDecimal | High precision and control over rounding modes. |
| printf | Direct and easy console formatting. |
| Math.floor/ceil | Always rounding down or up to a certain precision. |
| NumberFormat | Localization and configurable formatting. |

* **Escape Sequences in Java**

Escape sequences are special characters used in Java string literals to represent certain characters that are difficult to type directly. They begin with a backslash (\).

1. **\n – New Line**
   * **Purpose:** Moves the cursor to the next line.
   * **Example:** System.out.println("Hello\nWorld");
   * **Output:**

Hello

World

1. **\t – Tab**
   * **Purpose:** Inserts a horizontal tab.
   * **Example:** System.out.println("Hello\tWorld");
   * **Output:** Hello World
2. **\b – Backspace**
   * **Purpose:** Moves the cursor one position back (overwrites the last character).
   * **Example:** System.out.println("Hello\bWorld");
   * **Output:** HellWorld
3. **\r – Carriage Return**
   * **Purpose:** Moves the cursor to the beginning of the current line.
   * **Example:** System.out.println("Hello\rWorld");
   * **Output:** World (overwrites Hello)
4. **\\ – Backslash**
   * **Purpose:** Inserts a backslash (\) in the string.
   * **Example:** System.out.println("This is a backslash: \\");
   * **Output:** This is a backslash: \
5. **\' – Single Quote**
   * **Purpose:** Inserts a single quote (') in the string.
   * **Example:** System.out.println("It\'s a great day!");
   * **Output:** It's a great day!

* **User Input in Java**

In Java, user input is commonly taken through the Scanner class from the java.util package. The Scanner class allows us to read different types of input like strings, integers, floating-point numbers, and more.

**Steps to Take User Input:**

1. **Import the Scanner class**: To use the Scanner class, you need to import it from the java.util package.

import java.util.Scanner;

1. **Create a Scanner object**: Create an instance of the Scanner class to capture the input from the user.

Scanner scanner = new Scanner(System.in);

1. **Read different types of inputs**: The Scanner class has various methods to read different types of input. Here’s how to use them:

* **String Input**: Use nextLine() to read a line of text.

String name = scanner.nextLine(); // Reads a full line of text

* **Integer Input**: Use nextInt() to read an integer.

int age = scanner.nextInt(); // Reads an integer

* **Double Input**: Use nextDouble() to read a floating-point number.

double price = scanner.nextDouble(); // Reads a double

* **Boolean Input**: Use nextBoolean() to read a boolean value.

boolean isActive = scanner.nextBoolean(); // Reads a boolean (true/false)

* **Character Input**: For reading a single character, you can use next().charAt(0).

char gender = scanner.next().charAt(0); // Reads a single character

1. **Close the Scanner**: It’s a good practice to close the Scanner object after usage to free up resources.

**scanner.close();**

**Example Code:**

import java.util.Scanner;

public class UserInputExample {

public static void main(String[] args) {

// Create a Scanner object

Scanner scanner = new Scanner(System.in);

// Taking different inputs

System.out.print("Enter your name: ");

String name = scanner.nextLine();

System.out.print("Enter your age: ");

int age = scanner.nextInt();

System.out.print("Enter your salary: ");

double salary = scanner.nextDouble();

System.out.print("Are you active? (true/false): ");

boolean isActive = scanner.nextBoolean();

// Displaying the inputs

System.out.println("Name: " + name);

System.out.println("Age: " + age);

System.out.println("Salary: " + salary);

System.out.println("Active: " + isActive);

// Close the scanner

scanner.close();

}

}

**Short Notes (Important Points):**

* **Scanner Class**: Used for taking input in Java. It is a part of java.util package.
* **Methods for Input**:
  + nextLine(): Reads a full line (string).
  + nextInt(): Reads an integer.
  + nextDouble(): Reads a floating-point number.
  + nextBoolean(): Reads a boolean.
  + next().charAt(0): Reads a single character.
* **Closing Scanner**: Always close the Scanner object after use with scanner.close(); to avoid resource leaks.

Let me know if you need further details on any specific type of input!

* **Type Conversion and Casting in Java**

**1. Type Conversion** Type conversion is the process of converting one data type to another. In Java, there are two types of type conversion:

* **Implicit Conversion (Widening)**: Happens automatically when a smaller data type is assigned to a larger one.
  + Example: int to long, float to double.
  + **No data loss** as the target data type can hold larger values.

int num = 10;

long largerNum = num; // Implicit conversion from int to long

* **Explicit Conversion (Narrowing)**: Requires casting when converting a larger data type to a smaller one.
  + Example: double to int, long to short.
  + **Data loss** can occur since the smaller data type may not be able to hold the full value.

double num = 9.8;

int intNum = (int) num; // Explicit conversion (casting) from double to int

**2. Type Casting** Type casting refers to the process of converting one type to another using either implicit or explicit casting.

* **Widening Casting (Automatic)**: Automatically performed when converting from a smaller to a larger data type.

int a = 5;

double b = a; // Widening: int to double

* **Narrowing Casting (Explicit)**: Requires manual casting and can cause data loss.

double a = 9.78;

int b = (int) a; // Narrowing: double to int

**Summary:**

* **Implicit (Widening) Conversion**: Happens automatically from a smaller to a larger type.
* **Explicit (Narrowing) Conversion**: Requires manual casting and may cause data loss.
* Always be careful when narrowing types, as data may be truncated or lost.
* **Upcasting and Downcasting in Java**

In Java, **upcasting** and **downcasting** are used to cast objects between parent and child classes. These operations are fundamental in object-oriented programming when working with inheritance and polymorphism.

**1. What is Upcasting?**

**Definition:**  
Upcasting is the process of converting a child class reference into a parent class reference. It is **implicit** because a child class object is always compatible with a parent class reference.

**Key Points:**

* Upcasting happens automatically (implicitly).
* It allows a child class object to be treated as an instance of its parent class.
* Only the methods and properties defined in the parent class are accessible through the parent class reference.

**Example:**

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

class Dog extends Animal {

void bark() {

System.out.println("The dog barks.");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog(); // Child class object

Animal animal = dog; // Upcasting: Child to Parent

animal.eat(); // Allowed, as "eat()" is defined in Animal

// animal.bark(); // Error: Cannot call "bark()" using parent reference

}

}

**Explanation:**  
In this example:

* A Dog object is assigned to an Animal reference.
* The Animal reference can only access the eat() method, which is part of the Animal class.
* The bark() method, which belongs to the Dog class, cannot be accessed unless explicitly cast back to Dog (downcasting).

**2. What is Downcasting?**

**Definition:**  
Downcasting is the process of converting a parent class reference back into a child class reference. It must be done **explicitly** because the parent class reference does not always guarantee the presence of child-specific methods or fields.

**Key Points:**

* Downcasting requires explicit casting using parentheses.
* It can throw a **ClassCastException** at runtime if the reference is not of the correct type.
* Downcasting is typically used when a method returns a parent type, but the actual object is of a child type.

**Example:**

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

class Dog extends Animal {

void bark() {

System.out.println("The dog barks.");

}

}

public class Main {

public static void main(String[] args) {

Animal animal = new Dog(); // Upcasting

Dog dog = (Dog) animal; // Downcasting: Parent to Child

dog.bark(); // Allowed, as "animal" is actually a Dog object

}

}

**Explanation:**  
In this example:

* The animal reference initially refers to a Dog object.
* Using (Dog) animal, the parent reference is explicitly cast back to a child type (Dog).
* This allows access to the bark() method, which is specific to the Dog class.

**3. Why Do We Need Upcasting and Downcasting?**

**Use Cases for Upcasting:**

1. **Polymorphism:** Upcasting allows the use of a parent class reference to call overridden methods of the child class.

Animal animal = new Dog();

animal.eat(); // Calls the overridden eat() in Dog class

1. **Code Generalization:** It enables the use of the parent class reference for a group of child objects in a common way (e.g., storing different child objects in a single collection).

List<Animal> animals = new ArrayList<>();

animals.add(new Dog());

animals.add(new Cat());

**Use Cases for Downcasting:**

1. **Accessing Child-Specific Methods:** When the parent reference points to a child object, downcasting is required to access child-specific methods or properties.

Animal animal = new Dog();

if (animal instanceof Dog) {

Dog dog = (Dog) animal;

dog.bark();

}

**4. The Role of instanceof in Downcasting**

Before downcasting, it is a good practice to check the actual type of the object using the instanceof operator. This prevents ClassCastException at runtime.

**Example:**

Animal animal = new Dog();

if (animal instanceof Dog) {

Dog dog = (Dog) animal; // Safe downcasting

dog.bark();

} else {

System.out.println("The object is not a Dog.");

}

**5. What Happens When Downcasting Fails?**

If the object being cast is not of the child class type, a **ClassCastException** is thrown at runtime.

**Example:**

class Cat extends Animal {}

Animal animal = new Animal();

Dog dog = (Dog) animal; // Throws ClassCastException

**6. Practical Example: Using Upcasting and Downcasting Together**

**Scenario:**  
Suppose we have an Animal class with Dog and Cat as subclasses. A method processes a list of animals and performs actions specific to each animal type.

**Code:**

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

class Dog extends Animal {

void bark() {

System.out.println("The dog barks.");

}

}

class Cat extends Animal {

void meow() {

System.out.println("The cat meows.");

}

}

public class Main {

public static void main(String[] args) {

Animal animal1 = new Dog(); // Upcasting

Animal animal2 = new Cat(); // Upcasting

processAnimal(animal1);

processAnimal(animal2);

}

static void processAnimal(Animal animal) {

if (animal instanceof Dog) {

Dog dog = (Dog) animal; // Downcasting

dog.bark();

} else if (animal instanceof Cat) {

Cat cat = (Cat) animal; // Downcasting

cat.meow();

} else {

System.out.println("Unknown animal.");

}

}

}

**Output:**

The dog barks.

The cat meows.

**Explanation:**

1. The method processAnimal() uses instanceof to determine the type of the object.
2. Downcasting is performed safely to call child-specific methods (bark() for Dog and meow() for Cat).
3. **Advantages of Upcasting and Downcasting**

**Advantages of Upcasting:**

* Simplifies code by using parent class references.
* Enables polymorphism.
* Makes code flexible for future child class additions.

**Advantages of Downcasting:**

* Access child-specific properties or methods.
* Useful when working with generic methods that return parent types.

**Summary**

| **Concept** | **Upcasting** | **Downcasting** |
| --- | --- | --- |
| **Direction** | Child → Parent | Parent → Child |
| **Type of Casting** | Implicit | Explicit |
| **When to Use** | For polymorphism and generalization. | For accessing child-specific methods. |
| **Risk** | Safe | Can throw ClassCastException if invalid. |

* **Java Operators**

**1. Arithmetic Operators**

Used for basic mathematical calculations.

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

**2. Assignment Operators (Shorthand Operators)**

Assign values to variables.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| = | Assign | a = 10 |
| += | Add and assign | a += b (a = a + b) |
| -= | Subtract and assign | a -= b (a = a - b) |
| \*= | Multiply and assign | a \*= b (a = a \* b) |
| /= | Divide and assign | a /= b (a = a / b) |
| %= | Modulus and assign | a %= b (a = a % b) |

**3. Order of Operation**

1. **Parentheses**: ()
2. **Exponents**: ^
3. **Multiplication/Division/Modulus**: \*, /, %
4. **Addition/Subtraction**: +, -

**4. Shorthand Operators**

Simplify arithmetic operations.

* a += b is equivalent to a = a + b.
* a -= b is equivalent to a = a - b.
* Similar for \*, /, and %.

**5. Unary Operators**

Operate on a single operand.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| + | Unary plus (positive) | +a |
| - | Unary minus (negative) | -a |
| ++ | Increment (pre/post) | ++a or a++ |
| -- | Decrement (pre/post) | --a or a-- |
| ! | Logical NOT | !a |

**6. If-Else**

Used for decision-making.

if (condition) {

// Code block if condition is true

} else {

// Code block if condition is false

}

Example:

if (a > b) {

System.out.println("a is greater");

} else {

System.out.println("b is greater");

}

**7. Relational (Comparison) Operators**

Compare two values.

|  |  |  |
| --- | --- | --- |
| Operator | Description | 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 |

**8. Logical Operators in Java**

Logical operators are used to perform logical operations on two or more conditions. These operators return a boolean value (true or false).

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Description | Example |
| && | Logical AND | Returns true if both conditions are true. | (a > b) && (x < y) |
| || | Logical OR | If at least one condition is true, the result is true. | (a < b) || (b < 5) |
| ! | Logical NOT | Reverses the logical state of its operand (true to false and vice versa). | !(a > b) |

**9. Operator Precedence**

The order in which operators are evaluated.

|  |  |
| --- | --- |
| Precedence Level | Operators |
| **Highest** | Postfix: expr++, expr-- |
|  | Unary: ++expr, --expr, +, -, ~, ! |
|  | Arithmetic: \*, /, % |
|  | Addition: +, - |
|  | Shift: <<, >>, >>> |
|  | Relational: <, <=, >, >=, instanceof |
|  | Equality: ==, != |
|  | Bitwise AND: & |
|  | Bitwise XOR: ^ |
|  | Bitwise OR: ` |
|  | Logical AND: && |
|  | Logical OR: ` |
| **Lowest** | Assignment: =, +=, etc. |

**10. Intro to Number System**

Number systems represent values. Common types:

* **Decimal (Base 10)**: Numbers 0-9.
* **Binary (Base 2)**: Numbers 0 and 1.
* **Octal (Base 8)**: Numbers 0-7.
* **Hexadecimal (Base 16)**: Numbers 0-9 and A-F.

Example:

int decimal = 10; // Decimal

int binary = 0b1010; // Binary (prefix 0b)

int octal = 012; // Octal (prefix 0)

int hex = 0xA; // Hexadecimal (prefix 0x)

**11. Intro to Bitwise Operators**

Operate on individual bits.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| & | Bitwise AND | a & b |
| | | Bitwise OR | x | y |
| ^ | Bitwise XOR | a ^ b |
| ~ | Bitwise Complement | ~a |
| << | Left Shift | a << 2 |
| >> | Right Shift | a >> 2 |
| >>> | Unsigned Right Shift | a >>> 2 |

* **Bitwise Operators**

**1. AND (&)**

* **Rule:** 1 & 1 = 1, otherwise 0.
* **Example:**
  + **Scenario:** Two friends wear sunglasses on a sunny day. Only if **both** wear sunglasses can they see clearly.
  + **Binary:**

5 = 0101

3 = 0011

----------------

5 & 3 = 0001 (1 in decimal)

**2. OR (|)**

* **Rule:** 1 | 0 = 1; if **either** bit is 1, result is 1.
* **Example:**
  + **Scenario:** A TV can turn on with either the remote **or** the button.
  + **Binary:**

5 = 0101

3 = 0011

----------------

5 | 3 = 0111 (7 in decimal)

**3. XOR (^)**

* **Rule:** 1 ^ 1 = 0; 0 ^ 1 = 1; (Different = 1, Same = 0).
* **Example:**
  + **Scenario:** A couple decides to go for dinner only if **one** of them is free, but not both!
  + **Binary:**

5 = 0101

3 = 0011

----------------

5 ^ 3 = 0110 (6 in decimal)

**4. NOT (~)**

* **Rule:** Flips bits (1 becomes 0, and 0 becomes 1).
* **Example:**
  + **Scenario:** Your angry boss yells, "Do the opposite of everything!"
  + **Binary:**

5 = 0101

~5 = 1010 (-6 in decimal, because of 2's complement in signed integers)

**5. Left Shift (<<)**

* **Rule:** Moves bits to the left, adding 0s on the right (multiplies by 2).
* **Example:**
  + **Scenario:** Think of a conveyor belt shifting boxes to the left, doubling their quantity.
  + **Binary:**

5 = 0101

5 << 1 = 1010 (10 in decimal)

**6. Right Shift (>>)**

* **Rule:** Moves bits to the right, removing bits from the end (divides by 2).
* **Example:**
  + **Scenario:** Imagine downsizing a file by shifting its data bits right.
  + **Binary:**

5 = 0101

5 >> 1 = 0010 (2 in decimal)

**1. Simple if Statement**

* Executes a block of code only if the condition is true.

if (condition) {

// Code to execute if condition is true

}

**Example:**

if (x > 0) {

System.out.println("Positive number");

}

**2. if-else Statement**

* Executes one block of code if the condition is true, otherwise executes another block.

if (condition) {

// Code if condition is true

} else {

// Code if condition is false

}

**Example:**

if (x % 2 == 0) {

System.out.println("Even number");

} else {

System.out.println("Odd number");

}

**3. if-else if Ladder**

* Used to check multiple conditions. The first true condition block is executed.

if (condition1) {

// Code if condition1 is true

} else if (condition2) {

// Code if condition2 is true

} else {

// Code if none of the conditions are true

}

**Example:**

if (marks >= 90) {

System.out.println("Grade A");

} else if (marks >= 75) {

System.out.println("Grade B");

} else if (marks >= 50) {

System.out.println("Grade C");

} else {

System.out.println("Fail");

}

**4. Nested if Statements**

* An if statement inside another if or else block.

if (condition1) {

if (condition2) {

// Code if both conditions are true

}

}

**Example:**

if (age >= 18) {

if (hasVoterID) {

System.out.println("Eligible to vote");

} else {

System.out.println("Get a voter ID first");

}

}

**5. Ternary Operator (Shortcut for if-else)**

* A shorthand for simple if-else statements.

variable = (condition) ? valueIfTrue : valueIfFalse;

**Example:**

String result = (x > 0) ? "Positive" : "Negative";

System.out.println(result);

**6. Multiple Conditions with Logical Operators**

* Use logical operators (&&, ||) to combine conditions.

if (condition1 && condition2) {

// Code if both conditions are true

}

if (condition1 || condition2) {

// Code if at least one condition is true

}

**Example:**

if (age >= 18 && hasDrivingLicense) {

System.out.println("Eligible to drive");

}

if (day.equals("Saturday") || day.equals("Sunday")) {

System.out.println("It's a weekend");

}

**Key Points to Remember**

1. **Conditions** must always evaluate to a boolean (true or false).
2. Use braces {} for clarity, even if the block has one line.
3. Avoid deeply nested if statements for better readability.

* **Java Number System:**

In Java, the **Number System** defines how numbers are represented and processed. Java supports four types of number systems:

**1. Decimal (Base 10)**

* **Digits Used**: 0-9
* **Default system** in Java.
* **Example**: int a = 45;

**2. Binary (Base 2)**

* **Digits Used**: 0 and 1
* Represented with the **prefix 0b or 0B**.
* **Example**: int b = 0b101; // equals 5 in decimal

**3. Octal (Base 8)**

* **Digits Used**: 0-7
* Represented with the **prefix 0**.
* **Example**: int c = 071; // equals 57 in decimal

**4. Hexadecimal (Base 16)**

* **Digits Used**: 0-9 and A-F (or a-f) for values 10-15.
* Represented with the **prefix 0x or 0X**.
* **Example**: int d = 0x1A; // equals 26 in decimal

**Key Points to Remember**

**1. Automatic Conversion:**

Java converts binary, octal, or hexadecimal literals to decimal automatically.

* **Example**: System.out.println(0b110 + 0xA); // outputs 16

**2. Integer and Floating-Point Numbers:**

* **Integer Types**: byte, short, int, long.
* **Floating-Point Types**: float, double (for decimals).

**3. Wrapper Classes for Numbers:**

* Java provides classes like Integer, Double, etc., to handle numbers in an object-oriented way.

**4. Conversion Between Number Systems:**

Java provides built-in methods in the Integer class:

| **Method** | **Purpose** |
| --- | --- |
| Integer.toBinaryString() | Convert decimal to binary string |
| Integer.toOctalString() | Convert decimal to octal string |
| Integer.toHexString() | Convert decimal to hexadecimal string |
| Integer.parseInt(value, base) | Convert any base to decimal |

**Example Code**

public class NumberSystemDemo {

public static void main(String[] args) {

// Number literals

int decimal = 10;

int binary = 0b1010;

int octal = 012;

int hexadecimal = 0xA;

// Printing values

System.out.println("Decimal: " + decimal); // 10

System.out.println("Binary: " + binary); // 10

System.out.println("Octal: " + octal); // 10

System.out.println("Hexadecimal: " + hexadecimal); // 10

// Conversion examples

System.out.println("Decimal to Binary: " + Integer.toBinaryString(decimal)); // 1010

System.out.println("Binary to Decimal: " + Integer.parseInt("1010", 2)); // 10

}

}

**Short Notes for Quick Revision**

* **Prefixes for Number Systems**:
  + Binary: 0b
  + Octal: 0
  + Hexadecimal: 0x
* **Default Type for Numbers**:
  + int for integers (e.g., 5).
  + double for floating points (e.g., 5.0).
* **Common Conversion Methods**:
  + toBinaryString(), toOctalString(), toHexString(), parseInt().
* manually convert

**1. Decimal to Binary**

**Steps:**

1. Divide the decimal number by 2 (the base of binary).
2. Record the remainder (0 or 1).
3. Repeat the division with the quotient until the quotient becomes 0.
4. Write the remainders in **reverse order** (from bottom to top).

**Example: Convert 45 (decimal) to binary**

1. 45÷2=2245 \div 2 = 2245÷2=22 remainder **1**
2. 22÷2=1122 \div 2 = 1122÷2=11 remainder **0**
3. 11÷2=511 \div 2 = 511÷2=5 remainder **1**
4. 5÷2=25 \div 2 = 25÷2=2 remainder **1**
5. 2÷2=12 \div 2 = 12÷2=1 remainder **0**
6. 1÷2=01 \div 2 = 01÷2=0 remainder **1**

**Binary Result**: 45=10110145 = 10110145=101101 (read remainders from bottom to top)

**2. Binary to Decimal**

**Steps:**

1. Write down the binary number.
2. Multiply each binary digit by 2position2^{\text{position}}2position, starting from **0** on the right.
3. Add the results.

**Example: Convert 101101 (binary) to decimal**

1. 1×25=321 \times 2^5 = 321×25=32
2. 0×24=00 \times 2^4 = 00×24=0
3. 1×23=81 \times 2^3 = 81×23=8
4. 1×22=41 \times 2^2 = 41×22=4
5. 0×21=00 \times 2^1 = 00×21=0
6. 1×20=11 \times 2^0 = 11×20=1

**Decimal Result**: 32+0+8+4+0+1=4532 + 0 + 8 + 4 + 0 + 1 = 4532+0+8+4+0+1=45

**3. Decimal to Octal**

**Steps:**

1. Divide the decimal number by 8 (the base of octal).
2. Record the remainder (0-7).
3. Repeat the division with the quotient until the quotient becomes 0.
4. Write the remainders in **reverse order**.

**Example: Convert 45 (decimal) to octal**

1. 45÷8=545 \div 8 = 545÷8=5 remainder **5**
2. 5÷8=05 \div 8 = 05÷8=0 remainder **5**

**Octal Result**: 45=5545 = 5545=55

**4. Octal to Decimal**

**Steps:**

1. Write down the octal number.
2. Multiply each digit by 8position8^{\text{position}}8position, starting from **0** on the right.
3. Add the results.

**Example: Convert 55 (octal) to decimal**

1. 5×81=405 \times 8^1 = 405×81=40
2. 5×80=55 \times 8^0 = 55×80=5

**Decimal Result**: 40+5=4540 + 5 = 4540+5=45

**5. Decimal to Hexadecimal**

**Steps:**

1. Divide the decimal number by 16 (the base of hexadecimal).
2. Record the remainder (0-9, A-F for values 10-15).
3. Repeat the division with the quotient until the quotient becomes 0.
4. Write the remainders in **reverse order**.

**Example: Convert 45 (decimal) to hexadecimal**

1. 45÷16=245 \div 16 = 245÷16=2 remainder **13** (13 = D in hexadecimal)
2. 2÷16=02 \div 16 = 02÷16=0 remainder **2**

**Hexadecimal Result**: 45=2D45 = 2D45=2D

**6. Hexadecimal to Decimal**

**Steps:**

1. Write down the hexadecimal number.
2. Multiply each digit by 16position16^{\text{position}}16position, starting from **0** on the right.
3. For letters (A-F), use their decimal equivalents (A = 10, B = 11, ..., F = 15).
4. Add the results.

**Example: Convert 2D (hexadecimal) to decimal**

1. 2×161=322 \times 16^1 = 322×161=32
2. D×160=13D \times 16^0 = 13D×160=13 (D = 13 in decimal)

**Decimal Result**: 32+13=4532 + 13 = 4532+13=45

**Summary Table for Conversion**

|  |  |
| --- | --- |
| From/To | Steps |
| Decimal to Binary | Divide by 2, record remainders, reverse order. |
| Binary to Decimal | Multiply each digit by 2position2^{\text{position}}2position, add results. |
| Decimal to Octal | Divide by 8, record remainders, reverse order. |
| Octal to Decimal | Multiply each digit by 8position8^{\text{position}}8position, add results. |
| Decimal to Hex | Divide by 16, record remainders (0-9, A-F), reverse order. |
| Hex to Decimal | Multiply each digit by 16position16^{\text{position}}16position, add results. |

* **Java Comments (Short & Simple)**

Comments in Java are ignored by the compiler and are used to make code more readable or explain logic.

**Types of Comments**

1. **Single-line Comments**
   * Start with //.
   * Used for short notes on one line.

// This is a single-line comment

System.out.println("Hello, Java!"); // Print statement

1. **Multi-line Comments**
   * Start with /\* and end with \*/.
   * Used for commenting multiple lines.

/\*

This is a multi-line comment.

Useful for detailed explanations or disabling code.

\*/

1. **Documentation Comments**
   * Start with /\*\* and end with \*/.
   * Used to generate documentation with the javadoc tool.

/\*\*

\* This method prints a message to the console.

\*/

public void printMessage() {

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

}

**Best Practices**

* Explain **why** the code exists, not what it does.
* Keep comments meaningful and updated.
* Use sparingly; write self-explanatory code.

**Example:**

// Single-line comment

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

/\* Multi-line comment

explaining the code below \*/

int age = 25;

/\*\*

\* Prints a greeting message.

\*/

System.out.println("Welcome!");

* **loop**

A **loop** is a programming structure that allows a block of code to run multiple times based on a specified condition.

1. **Code that runs multiple times based on a condition:** A loop executes a set of instructions repeatedly as long as the condition remains true.
2. **Repeated execution of code:** Loops help automate repetitive tasks by executing the same code multiple times without needing to write it again.
3. **Loops automate repetitive tasks:** Loops save time and effort by handling repetitive actions, such as processing lists or calculating sums.
4. **Types of Loops:**
   * **While loop:** Executes as long as the condition is true.
   * **For loop:** Runs a specified number of times, usually with a counter.
   * **Do-while loop:** Executes at least once and continues as long as the condition is true after each iteration.
5. **Iterations:** The number of times a loop runs. For example, if a loop executes 5 times, it has 5 iterations.

**While Loop**

A **while loop** is used to repeat a block of code as long as a specified condition remains true.

**Key Points:**

1. **Iterations:**
   * It defines the number of times the loop will run, depending on the condition.
   * The loop continues executing until the condition becomes **false**.
2. **Non-standard Conditions:**
   * Used when you don’t know exactly how many times the loop will run (i.e., the loop condition might be based on a dynamic or non-fixed condition).
   * Example: Keep asking for user input until a valid input is given.
3. **Repeating a Block of Code:**
   * The code inside the loop is executed repeatedly as long as the condition is true.
   * Example: while (x < 5) { x++; } will keep increasing x until it reaches 5.
4. **Update Condition:**
   * It’s essential to update the condition in the loop (e.g., increment or decrement a counter variable).
   * **If not updated**, the loop will run indefinitely, creating an **infinite loop**. This can freeze or crash your program.

**Example:**

int x = 0;

while (x < 5) {

System.out.println(x);

x++; // This is the update step

}

In the example, x starts at 0, and the loop continues until x reaches 5. The update (x++) ensures the loop stops.

**Important Points:**

* Always ensure the loop condition will eventually become false.
* Use it when you are uncertain about the exact number of iterations needed.
* **For Loop**

The **for loop** in Java is used when you know the exact number of iterations or when you want to iterate over a range of values.

**Syntax:**

for (initialization; condition; increment/decrement) {

// Code to be executed in each iteration

}

* **Initialization**: This step is executed once before the loop starts. It’s used to initialize the loop control variable.
* **Condition**: This condition is checked before each iteration. If it's true, the loop will continue; if false, the loop stops.
* **Increment/Decrement**: After each iteration, the loop control variable is updated.

**Example:**

public class Main {

public static void main(String[] args) {

for (int i = 0; i < 5; i++) {

System.out.println(i); // Prints 0 to 4

}

}

}

**Explanation**:

* int i = 0: Initializes the counter i to 0.
* i < 5: This is the condition; the loop runs as long as i is less than 5.
* i++: Increments the counter by 1 after each iteration.

**Important Points:**

* The loop will run 5 times (from i = 0 to i = 4).
* **Initialization**, **Condition**, and **Increment/Decrement** are clearly defined within the loop's syntax.
* **Do-While Loop**

The **do-while loop** ensures that the block of code inside the loop will be executed **at least once**, and then it will continue to execute as long as the condition is true.

**Syntax:**

do {

// Code to be executed

} while (condition);

* The code inside the do block is executed **first**, then the condition is checked.
* If the condition is true, the loop runs again; if false, it stops.

**Example:**

public class Main {

public static void main(String[] args) {

int i = 0;

do {

System.out.println(i); // Prints 0 to 4

i++;

} while (i < 5);

}

}

**Explanation**:

* The code System.out.println(i) runs at least once before checking the condition.
* i++ increments the value of i after each iteration.
* The loop will run as long as i < 5.

**Important Points:**

* The **do-while loop guarantees at least one iteration**, even if the condition is false initially.
* The **condition is checked after** the execution of the loop body, not before.

**Key Differences Between for and do-while in Java:**

|  |  |  |
| --- | --- | --- |
| Feature | For Loop | Do-While Loop |
| Initialization | Happens once before the loop starts. | Happens only once, just before the loop starts. |
| Condition Check | Condition is checked **before** the loop starts. | Condition is checked **after** the loop runs. |
| Guaranteed Execution | May not execute if the condition is initially false. | Will execute **at least once**, even if the condition is false. |
| Use Case | Ideal when the number of iterations is known in advance. | Ideal when you need the loop to run at least once. |

**Functions / Methods:**

Functions (or methods in some programming languages) are essential building blocks of programming. They help us write organized, reusable, and efficient code. Let's break down the points step by step:

**1. Definition: Blocks of Reusable Code**

* **What is a Function?** A function is a **block of code** designed to perform a specific task.
* Instead of writing the same code multiple times, you write it once inside a function and "call" the function whenever needed.
* Think of a function like a machine: You input something, it processes it, and gives you an output.

**Example in Java:**

public static int addNumbers(int a, int b) {

return a + b; // Adds two numbers and returns the result

}

public static void main(String[] args) {

System.out.println(addNumbers(5, 10)); // Output: 15

}

**2. DRY Principle: "Don't Repeat Yourself"**

* **What is DRY?** DRY stands for "**Don't Repeat Yourself**," a programming principle that encourages **code reusability** and avoids redundancy.
* Instead of writing the same code multiple times, you write it once in a function and reuse it whenever necessary.
* Benefits of DRY:
  + Saves time and effort.
  + Makes the code easier to maintain and debug.
  + Reduces errors since a single change in the function updates all its uses.

**Example Without DRY:**

System.out.println(5 + 10);

System.out.println(7 + 8);

System.out.println(12 + 3);

**Example Using DRY with Functions:**

public static int addNumbers(int a, int b) {

return a + b;

}

public static void main(String[] args) {

System.out.println(addNumbers(5, 10));

System.out.println(addNumbers(7, 8));

System.out.println(addNumbers(12, 3));

}

**3. Usage: Organizes Code and Performs Specific Tasks**

* Functions **organize code** into logical blocks, making it easier to understand, maintain, and reuse.
* Each function is responsible for a **specific task**. For example:
  + A function can calculate the area of a circle.
  + Another function can format and print a message.
* Functions make programs **modular**, meaning each function can be independently tested or replaced without affecting the rest of the code.

**Example of Organizing Code:**

public static int calculateArea(int length, int width) {

return length \* width; // Calculates area of a rectangle

}

public static void printArea(int area) {

System.out.println("The area is: " + area);

}

public static void main(String[] args) {

int area = calculateArea(5, 10);

printArea(area); // Organized into clear tasks

}

**4. Naming Rules: Same as Variable Names (camelCase)**

* Functions follow **naming rules** similar to variables:
  + Can contain letters, numbers, and underscores (\_), but cannot start with a number.
  + Cannot use special characters or spaces.
  + Should not be a reserved keyword like class, return, etc.
* Use **camelCase** for naming functions:
  + First word starts with a lowercase letter.
  + Each subsequent word starts with an uppercase letter.
  + Example: calculateArea, addNumbers.

**Good Naming Practice:**

* Use **descriptive names** that explain what the function does.
* Avoid generic names like doSomething or temp.

**Example of Proper Naming:**

public static int calculateSum(int a, int b) {

return a + b; // Descriptive name, clear purpose

}

**Example of Improper Naming:**

public static int x(int a, int b) {

return a + b; // Unclear name, hard to understand purpose

}

**Summary:**

* **Functions** make code reusable and organized.
* The **DRY principle** ensures you don't repeat code unnecessarily.
* Functions perform specific tasks, making programs modular and easy to maintain.
* Follow proper **naming rules** (camelCase) to create meaningful and understandable function names.

**Method Syntax in Java**

1. **Follows the same rules as variable names**
   * The name of a method should follow the same rules as naming variables.
   * It can contain letters, numbers, underscores (\_), or dollar signs ($).
   * It cannot start with a number and must not use special characters or spaces.
   * Example: Valid method names include calculateSum, displayMessage, and find\_Max.
2. **Use () to contain parameters**
   * The parentheses () after the method name are where you define parameters.
   * **Parameters** are variables that the method takes as input to perform its task.
   * If a method doesn’t need input, you leave the parentheses empty.
   * Example:

void greet() { // no parameters

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

}

void addNumbers(int a, int b) { // parameters: a and b

System.out.println(a + b);

}

1. **Invoke (call) by using the method name followed by ()**
   * To use a method, you call it by its name followed by () and pass arguments (if the method expects parameters).
   * Example:

greet(); // calls the greet method

addNumbers(5, 10); // calls addNumbers with 5 and 10 as arguments

1. **Fundamental for code organization and reusability**
   * Methods are like reusable pieces of code that let you avoid writing the same code multiple times.
   * They make your code cleaner and more organized by splitting tasks into small, manageable parts.
   * Example:  
     Instead of writing the same code to calculate the square of a number multiple times, you can use a method:

int square(int number) {

return number \* number;

}

// You can call this method wherever needed:

System.out.println(square(4)); // Output: 16

System.out.println(square(7)); // Output: 49

**Why Are Methods Important?**

* **Reusability:** You can call the same method multiple times instead of repeating the code.
* **Code readability:** Methods make your code easier to read and understand by dividing it into logical blocks.
* **Maintainability:** Fixing or updating a method automatically updates all its usages.
* **Flexibility:** You can pass different inputs (arguments) to methods to perform a variety of tasks.

**Quick Summary of Syntax:**

returnType methodName(parameters) {

// Method body (what the method does)

}

Example:

int multiply(int a, int b) { // Method with two parameters

return a \* b; // Returns the result

}

To use it:

System.out.println(multiply(3, 4)); // Output: 12

**Return Statement**

1. **Definition**:  
   The return statement is used to send a value from a function back to the part of the program where the function was called.
   * **Analogy**: Think of it as saying, *"Ek glass paani laao."* The function (person you asked) fetches the glass of water and gives it back to you (returns the result).
2. **What Can Be Returned**:  
   A function can return:
   * **A specific value**: e.g., a number, string, etc.
   * **A variable**: A pre-defined value stored in a variable.
   * **A calculation**: Directly return the result of an expression or operation.
3. **Ends the Function**:  
   Once the return statement is executed, the function immediately stops executing further code, and control is passed back to the caller.
4. **Function Call Analogy**:  
   A function call is like delegating a task. The code “jumps” to the function to complete the task and then “jumps back” with the result.  
   Example:
   * You call a bakery (function) to bake a cake (task).
   * The bakery completes the cake and gives it back to you (return value).
5. **Why It's Useful**:
   * Simplifies code by avoiding repetition.
   * Allows modular, reusable, and testable functions.

**Examples in Code**

**Example 1: Returning a Value**

int addNumbers(int a, int b) {

return a + b; // Returns the sum of a and b

}

public static void main(String[] args) {

int result = addNumbers(5, 7); // Function call

System.out.println("The sum is: " + result); // Prints "The sum is: 12"

}

**Example 2: Return Ends the Function**

String greet(String name) {

if (name == null) {

return "Hello, Guest!"; // Function stops here if name is null

}

return "Hello, " + name; // If name is not null, this line executes

}

public static void main(String[] args) {

System.out.println(greet("John")); // Output: Hello, John

System.out.println(greet(null)); // Output: Hello, Guest!

}

**Example 3: Returning a Calculation**

double calculateArea(double radius) {

return 3.14 \* radius \* radius; // Returns the area of the circle

}

public static void main(String[] args) {

double area = calculateArea(5);

System.out.println("Area of the circle: " + area); // Prints "Area of the circle: 78.5"

}

**Example 4: No Return (Void Functions)**

If you don’t need to return a value, you can use void functions. However, these do not use return with a value.

void printMessage() {

System.out.println("This is a message!"); // Simply prints the message

}

public static void main(String[] args) {

printMessage(); // Output: This is a message!

}

**Key Points to Remember**:

* Functions with return must specify a return type (e.g., int, String, double) in the method signature.
* Use void if no value is returned.
* The return keyword ensures the function sends a result back to the caller.

**1. Input Values That a Function Takes**

* Parameters are the placeholders in a function definition that specify the type of data the function expects as input.
* Arguments are the actual values you pass into the function when calling it.

**Example:**

void greet(String name) { // 'name' is a parameter

System.out.println("Hello, " + name);

}

greet("John"); // "John" is the argument

**2. Parameters Put Value In, While Return Gets Value Out**

* **Parameters** bring values **into** the function for processing.
* The **return statement** sends values **out** of the function back to the caller.

**Example:**

int add(int a, int b) { // 'a' and 'b' are parameters

return a + b; // Returns the sum of a and b

}

int sum = add(5, 3); // Passing 5 and 3 as arguments

System.out.println(sum); // Outputs: 8

**3. Example: "Ek Packet Dahi Laao"**

* Imagine you're giving an instruction: "Bring one packet of curd."
* **Parameter**: The instruction (e.g., *bring one packet of something*).
* **Argument**: The specific item you bring (e.g., *Amul Curd*).

**Java Analogy:**

void bringItem(String item) { // 'item' is the parameter

System.out.println("Bringing: " + item);

}

bringItem("Amul Curd"); // "Amul Curd" is the argument

**4. Naming Convention: Same as Variable Names**

* Parameters follow Java's variable naming rules:
  + Start with a letter or \_.
  + Use meaningful names that describe their purpose.
  + Avoid special characters and reserved keywords.

**Example:**

void calculateArea(int width, int height) { // Parameters with meaningful names

System.out.println("Area: " + (width \* height));

}

**5. Parameter vs Argument**

* **Parameter**: Placeholder variable defined in the function signature.
* **Argument**: Actual value you pass into the function.

**Example:**

void displayMessage(String message) { // 'message' is a parameter

System.out.println(message);

}

displayMessage("Hello, World!"); // "Hello, World!" is the argument

**6. Examples: Functions We Have Already Used**

* **System.out.print()**:

System.out.print("Hello"); // "Hello" is the argument

* **Scanner.nextInt()**:

Scanner sc = new Scanner(System.in);

int number = sc.nextInt(); // Reads user input; no arguments passed

**7. Multiple Parameters**

* A function can take more than one parameter. When calling the function, you must pass arguments for all parameters in the correct order.

**Example:**

void printDetails(String name, int age, String city) {

System.out.println("Name: " + name + ", Age: " + age + ", City: " + city);

}

printDetails("Alice", 25, "Mumbai"); // Passing three arguments

**8. Default Values in Java (Using Method Overloading)**

* Java does not support default parameter values directly. Instead, you can use **method overloading** to achieve the same effect.

**Example:**

// Overloaded methods to handle default values

void greet(String name) {

System.out.println("Hello, " + name);

}

void greet() { // Method without arguments (default behavior)

greet("Guest"); // Calls the other method with a default value

}

public static void main(String[] args) {

Main obj = new Main();

obj.greet(); // Outputs: Hello, Guest

obj.greet("John"); // Outputs: Hello, John

}

**What is an Array in Java?**

An **Array** is a data structure in Java used to store multiple values of the same data type in a single variable.

1. **Definition**:
   * An array is a container object that holds a fixed number of values of a single type.
   * Each item in an array is called an **element**, and each element is accessed using an **index**.
2. **Key Features**:
   * Arrays **start indexing from 0**.
   * Arrays are **fixed in size**, meaning the size must be specified when it is created and cannot be changed afterward.
   * Arrays can store **primitive data types** (like int, double) or **objects** (like String, CustomClass).

**Syntax of Array in Java**

1. **Declaring an Array**:

dataType[] arrayName; // Preferred syntax

// OR

dataType arrayName[];

1. **Creating an Array**:

arrayName = new dataType[size];

1. **Declaring and Creating in One Step**:

dataType[] arrayName = new dataType[size];

1. **Initializing an Array**:

int[] numbers = {10, 20, 30, 40}; // Direct initialization

1. **Accessing Array Elements**:

arrayName[index]; // Access an element by index

1. **Finding Array Length**:

arrayName.length; // Returns the size of the array

**Example**

public class ArrayExample {

public static void main(String[] args) {

// Declare and initialize an array

int[] numbers = {1, 2, 3, 4, 5};

// Accessing array elements

System.out.println("First element: " + numbers[0]);

System.out.println("Second element: " + numbers[1]);

// Find the length of the array

System.out.println("Array length: " + numbers.length);

}

}

**Types of Arrays in Java**

1. **Single-Dimensional Array**:
   * A list of values in a single row.
   * Example:

int[] array = {1, 2, 3, 4};

1. **Two-Dimensional Array (2D Array)**:
   * Represents a table or matrix with rows and columns.
   * Example:

int[][] matrix = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

1. **Multi-Dimensional Arrays**:
   * Arrays with more than two dimensions.
   * Example (3D Array):

int[][][] cube = new int[2][3][4];

**Working with 2D Arrays**

1. **Declaration and Initialization**:

int[][] matrix = new int[3][3];

1. **Assigning Values**:

matrix[0][0] = 1;

matrix[0][1] = 2;

matrix[1][0] = 3;

1. **Iterating Over 2D Arrays**:

for (int i = 0; i < matrix.length; i++) {

for (int j = 0; j < matrix[i].length; j++) {

System.out.print(matrix[i][j] + " ");

}

System.out.println();

}

**Important Points About Arrays**

1. **Advantages**:
   * Arrays allow random access using indices.
   * They are memory-efficient.
2. **Limitations**:
   * Fixed size: Cannot grow or shrink dynamically.
   * Homogeneous data: Can only store elements of the same type.
3. **Commonly Used Methods**:
   * **Iterate**: Using loops (e.g., for, enhanced for).
   * **Copy**: Use System.arraycopy or Arrays.copyOf.
4. **Multi-line Declaration**:

int[] numbers = {

10, 20, 30,

40, 50

};

* **Process vs Object Oriented**

**1. Basic Concept**

* **Process-Oriented Programming** focuses on **functions** or **procedures** that operate on data. The program is designed around a sequence of steps (processes or functions).
* **Object-Oriented Programming (OOP)** focuses on **objects** that contain both **data** and **functions**. The program is structured around objects that interact with each other.

**2. Approach**

* **Process-Oriented Programming** follows a **top-down approach**, where the main process is broken down into smaller functions.
* **Object-Oriented Programming** follows a **bottom-up approach**, where small, reusable objects are defined first and then integrated to create larger systems.

**3. Focus**

* **Process-Oriented** focuses on **what tasks** need to be performed and organizes them into functions.
* **OOP** focuses on **what things** (objects) need to be modeled and defines their behavior and state.

**4. Data and Function**

* **Process-Oriented** keeps **data separate** from functions. Functions act on data passed to them.
* **OOP** encapsulates **data and functions together** into objects, which makes the data more secure and prevents accidental modification from outside the object.

**5. Modularity**

* **Process-Oriented** programs are typically **less modular**. Since functions are separate from data, changes in data structure may require changes in many functions.
* **OOP** programs are highly **modular**. Since each object is self-contained, changes to one object don’t necessarily affect other parts of the system.

**6. Reusability**

* **Process-Oriented** offers limited **reusability** of functions, as they are usually designed for specific tasks.
* **OOP** enhances **reusability** because objects can be reused in different scenarios, and inheritance allows new objects to be created based on existing ones.

**7. Extensibility**

* **Process-Oriented** systems can be **difficult to extend**. Adding new functionality might require changes to the entire flow of the process.
* **OOP** systems are **easier to extend**. New objects or functionalities can be added without disturbing the existing code.

**8. State**

* **Process-Oriented** systems have functions that manipulate global or passed data, but there’s no internal state management for objects.
* **OOP** provides **state management** inside objects. Each object has its own state, and its behavior is defined by methods acting on that state.

**9. Examples**

* **Process-Oriented Programming**: C, older procedural languages.
* **OOP**: Java, C++, Python, C#, Ruby, etc.

**10. Real-World Analogy**

* **Process-Oriented**: Think of a **restaurant** where you give an order to a chef who performs the cooking (a function) and gives you your food (data). The process is the most important thing.
* **OOP**: Think of a **robot** that can perform multiple tasks (like cooking, cleaning, and helping people). The robot itself is an **object** with various capabilities (methods) and internal properties (state).

**Key Differences:**

| **Feature** | **Process-Oriented Programming** | **Object-Oriented Programming** |
| --- | --- | --- |
| **Main Focus** | Functions and processes | Objects and their interactions |
| **Data** | Separate from functions | Encapsulated with functions |
| **Modularity** | Low | High |
| **Reusability** | Low | High |
| **Extensibility** | Low | High |
| **Programming Style** | Top-down | Bottom-up |
| **State Management** | External data management | Internal state management |

**Summary:**

* **Process-Oriented Programming** is suitable for simpler tasks where the flow of execution is straightforward.
* **Object-Oriented Programming** is better for complex systems that require modularity, reusability, and maintainability.
* **Instance Variables and Methods (Explanation)**

In Java, **instance variables** and **instance methods** are crucial concepts used to define the behavior and state of objects in a class.

**1. Instance Variables:**

* **Definition**: Instance variables are variables that are defined inside a class, but outside any methods or constructors. These variables represent the state or properties of an object.
* **Key Points**:
  + **Unique to each object**: Every object (instance of a class) will have its own copy of instance variables. This means that each object can have different values for these variables.
  + **Declared inside class**: Instance variables are declared within the class but outside any method or constructor.
  + **Default values**: If not explicitly initialized, instance variables get default values based on their data types:
    - int, float, double, long → 0
    - boolean → false
    - Object reference → null
  + **Access Modifiers**: Instance variables can be declared with various access modifiers such as public, private, or protected. The access level determines the visibility of the variable.
* **Example**:

class Car {

// Instance variables

String model;

int year;

// Constructor to initialize instance variables

Car(String model, int year) {

this.model = model;

this.year = year;

}

}

public class Main {

public static void main(String[] args) {

// Creating objects of class Car

Car car1 = new Car("Toyota", 2020);

Car car2 = new Car("Honda", 2022);

// Accessing instance variables

System.out.println(car1.model + " " + car1.year); // Output: Toyota 2020

System.out.println(car2.model + " " + car2.year); // Output: Honda 2022

}

}

**2. Instance Methods:**

* **Definition**: Instance methods are functions or behaviors that are associated with an object (instance of a class). They operate on the instance variables of the object and define what the object can do.
* **Key Points**:
  + **Belongs to the object**: Instance methods are called on instances (objects) of the class.
  + **Can access instance variables**: These methods can access and modify the values of instance variables of the object they belong to.
  + **Default Access Modifier**: If no access modifier is specified, instance methods have default package-level visibility.
  + **Called using an object**: Instance methods are typically called using the object of the class (e.g., car1.methodName()).
* **Example**:

class Car {

// Instance variables

String model;

int year;

// Constructor to initialize instance variables

Car(String model, int year) {

this.model = model;

this.year = year;

}

// Instance method

void displayDetails() {

System.out.println("Model: " + model + ", Year: " + year);

}

}

public class Main {

public static void main(String[] args) {

// Creating objects of class Car

Car car1 = new Car("Toyota", 2020);

Car car2 = new Car("Honda", 2022);

// Calling instance method

car1.displayDetails(); // Output: Model: Toyota, Year: 2020

car2.displayDetails(); // Output: Model: Honda, Year: 2022

}

}

**Summary of Important Points:**

**1. Instance Variables:**

* **Declare the state of the object**: They hold the properties or data that define the state of an object.
* **Unique for each object**: Each object gets its own copy of the instance variables.
* **Accessed via objects**: Instance variables are accessed using the object, like object.variable.
* **Default values**: If not explicitly initialized, instance variables get default values (e.g., 0 for numbers, false for booleans, null for objects).
* **Scope**: Instance variables are accessible throughout the class and any instance methods of the class.
* **Lifetime**: They exist as long as the object exists, and are destroyed when the object is destroyed.
* **Can be modified**: Instance variables can be modified within instance methods.

**2. Instance Methods:**

* **Define behavior (functions) of the object**: Instance methods define the actions that objects of the class can perform.
* **Can access and modify instance variables**: Instance methods can read and change the values of instance variables.
* **Called using object references**: Instance methods are called using an object, like object.method().
* **Can have parameters and return values**: Instance methods can take inputs (parameters) and return outputs (return values).
* **Scope**: Instance methods have access to instance variables and can call other instance methods in the class.
* **Lifetime**: Instance methods exist as long as the object exists and are invoked as needed.
* **Encapsulation**: Instance methods and variables together help in encapsulating the behavior and state of an object.
* **Strings in Java**

**Part 1: Introduction to Strings**

1. **What is a String?**
   * A string in Java is a sequence of characters. It is immutable (cannot be changed once created).

**Example:**

String str = "Hello, World!";

**Part 2: String Initialization**

1. **String Initialization via Literals**
   * Strings created using literals are stored in a string pool to optimize memory.

**Example:**

String str1 = "Hello";

1. **String Initialization with new Keyword**
   * Using new creates a string object in heap memory, even if the string exists in the pool.

**Example:**

String str2 = new String("Hello");

**Part 3: Basic String Methods**

1. **length()**
   * Returns the length of the string.

**Example:**

String str = "Hello, World!";

int len = str.length(); // 13

1. **charAt(int index)**
   * Returns the character at a specific index in the string.

**Example:**

char ch = str.charAt(0); // 'H'

1. **substring(int beginIndex, int endIndex)**
   * Returns a substring from the given start and end indices.

**Example:**

String substr = str.substring(7, 12); // "World"

1. **equals(Object obj)**
   * Compares the string with another object to check equality (content comparison).

**Example:**

String str1 = "Hello";

String str2 = "Hello";

boolean isEqual = str1.equals(str2); // true

1. **toLowerCase() and toUpperCase()**
   * Converts the string to lowercase or uppercase.

**Example:**

String lower = str.toLowerCase(); // "hello, world!"

String upper = str.toUpperCase(); // "HELLO, WORLD!"

**Part 4: String Concatenation**

1. **Concatenation using + Operator**
   * Joins two or more strings together.

**Example:**

String str1 = "Hello";

String str2 = "World";

String result = str1 + " " + str2; // "Hello World"

1. **Concatenation using concat() Method**
   * Joins two strings using the concat() method.

**Example:**

String result = str1.concat(" ").concat(str2); // "Hello World"

**Part 5: StringBuilder and StringBuffer**

1. **StringBuilder and StringBuffer**
   * These classes allow modification of strings, as they are mutable.
   * StringBuilder is faster as it is not synchronized.
   * StringBuffer is thread-safe but slower.

**Example with StringBuilder:**

StringBuilder sb = new StringBuilder();

sb.append("Hello");

sb.append(" ");

sb.append("World");

String result = sb.toString(); // "Hello World"

**Part 6: String Comparison**

1. **String Comparison with equals()**
   * Compares two strings for equality by content, not reference.

**Example:**

String str1 = "Hello";

String str2 = new String("Hello");

boolean isEqual = str1.equals(str2); // true

1. **String Comparison with ==**
   * The == operator compares references (memory locations), not content.

**Example:**

System.out.println(str1 == str2); // false (different memory locations)

**Part 7: String Formatting**

1. **String Formatting with String.format()**
   * Formats strings using placeholders (%s, %d, etc.).

**Example:**

String formatted = String.format("My name is %s and I am %d years old", "Amol", 25);

System.out.println(formatted); // "My name is Amol and I am 25 years old"

1. **String Formatting with System.out.printf()**
   * Prints a formatted string directly to the console.

**Example:**

System.out.printf("My name is %s and I am %d years old", "Amol", 25);

**Part 8: Regular Expressions with Strings**

1. **String Matching with matches()**
   * Checks if a string matches a regular expression.

**Example:**

String str = "123-45-6789";

boolean isValid = str.matches("\\d{3}-\\d{2}-\\d{4}"); // Valid SSN format

1. **Using replaceAll() for String Manipulation**
   * Replaces parts of the string based on a regular expression.

**Example:**

String modifiedStr = str.replaceAll("\\d", "X"); // "XXX-XX-XXXX"

**Part 9: String.join() Method (Java 8 and later)**

1. **Joining Multiple Strings**
   * Joins multiple strings with a delimiter.

**Example:**

String result = String.join(", ", "apple", "banana", "cherry");

System.out.println(result); // "apple, banana, cherry"

**Part 10: String Pool and Interning**

1. **String Pool (String Literal Pool)**
   * A pool where string literals are stored to save memory and prevent duplicate objects.

**Example:**

String str1 = "Hello"; // String pool

String str2 = new String("Hello"); // Heap memory, not in pool

1. **String Interning**
   * Interning a string means that a string is stored in the pool, even if it was created using new.

**Example:**

String str2 = new String("Hello").intern(); // Interns the string

System.out.println(str1 == str2); // true

**Part 11: Advanced String Methods**

1. **contains()**
   * Checks if a string contains a specified sequence of characters.

**Example:**

boolean contains = str.contains("World"); // true

1. **indexOf()**
   * Returns the index of the first occurrence of a character or substring.

**Example:**

int index = str.indexOf("World"); // 7

1. **split()**
   * Splits a string into an array of substrings based on a delimiter.

**Example:**

String[] words = str.split(", "); // ["Hello", "World!"]

**Ways to Declare an Object in Java**

**1. Using new Keyword (Most Common)**

class Car {

void display() {

System.out.println("Car is running");

}

}

public class Main {

public static void main(String[] args) {

Car myCar = new Car(); // Object creation using new keyword

myCar.display();

}

}

**Important Points:**

* Calls the constructor of the class.
* Allocates memory dynamically in the heap.
* Default constructor is invoked if no constructor is defined.

**2. Using Factory Methods (Static Method)**

class Bike {

static Bike getInstance() {

return new Bike(); // Factory method returning new object

}

void display() {

System.out.println("Bike is ready");

}

}

public class Main {

public static void main(String[] args) {

Bike myBike = Bike.getInstance();

myBike.display();

}

}

**Important Points:**

* Used when object creation logic needs to be abstracted.
* Common in Singleton and Factory Design Patterns

1. **Using Class.forName() (Reflection)**

class Mobile {

void show() {

System.out.println("Mobile is working");

}

}

public class Main {

public static void main(String[] args) throws Exception {

Class<?> cls = Class.forName("Mobile");

Mobile myMobile = (Mobile) cls.getDeclaredConstructor().newInstance();

myMobile.show();

}

}

**Important Points:**

* Dynamically loads class and creates an instance.
* Used in JDBC and frameworks like Spring & Hibernate.

1. **Using clone() Method**

class Person implements Cloneable {

String name = "John";

public Object clone() throws CloneNotSupportedException {

return super.clone();

}

}

public class Main {

public static void main(String[] args) throws CloneNotSupportedException {

Person p1 = new Person();

Person p2 = (Person) p1.clone(); // Creating object using clone

System.out.println(p2.name);

}

}

**Important Points:**

* Requires Cloneable interface to avoid CloneNotSupportedException.
* Creates a copy of the existing object.
* Performs shallow copy unless manually overridden for deep copy.

1. **Using Object Deserialization**

import java.io.\*;

class Student implements Serializable {

String name;

Student(String name) {

this.name = name;

}

}

public class Main {

public static void main(String[] args) throws Exception {

// Serialization

Student s1 = new Student("Alice");

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("student.ser"));

out.writeObject(s1);

out.close();

// Deserialization

ObjectInputStream in = new ObjectInputStream(new FileInputStream("student.ser"));

Student s2 = (Student) in.readObject();

in.close();

System.out.println("Deserialized Student Name: " + s2.name);

}

}

**Important Points:**

* Serializable interface is required.
* Used when persisting objects to a file or database.
* Can reconstruct an object from its serialized state.

1. **Using Anonymous Inner Class**

abstract class Animal {

abstract void sound();

}

public class Main {

public static void main(String[] args) {

Animal obj = new Animal() {

void sound() {

System.out.println("Roar!");

}

};

obj.sound();

}

}

**Important Points:**

* Creates an unnamed object.
* Common in GUI programming (e.g., ActionListener in Swing).
* Can override methods inline.

1. **Using Enum (Singleton)**

enum Singleton {

INSTANCE;

void showMessage() {

System.out.println("Singleton using Enum");

}

}

public class Main {

public static void main(String[] args) {

Singleton.INSTANCE.showMessage();

}

}

**Important Points:**

* Best way to implement Singleton in Java.
* Thread-safe and prevents multiple instantiations.

**Important Points About Objects in Java**

1. **Heap Memory** - Objects are stored in heap memory, and references are stored in stack memory.
2. **Garbage Collection** - Unused objects are collected by the JVM garbage collector.
3. **Reference vs Primitive** - Objects are reference types, whereas primitive types (int, double, etc.) are stored directly in stack memory.
4. **Pass-by-Reference vs Pass-by-Value** - In Java, objects are passed by reference (internally, it's a copy of the reference).
5. **this Keyword** - Used to refer to the current object's instance variables or methods.
6. **super Keyword** - Used to refer to the parent class’s members.
7. **Immutable vs Mutable Objects** - String is immutable, whereas ArrayList is mutable.

**1. Folder Structure**

* Your project should be structured as:

JavaProject

├── src

├── Day\_5\_Pratic

├── CarMain.java

├── Day\_6\_practice

├── Car.java

├── ObjectPractice.java

**2. Code Structure**

* **ObjectPractice.java** (Day\_6\_practice package):

package Day\_6\_practice;

import Day\_5\_Pratic.CarMain;

public class ObjectPractice {

public static void main(String[] args) {

CarMain obj2 = new CarMain(); // Create an object of CarMain class

System.out.println("Name: " + obj2.name2); // Access public variable

obj2.myWork2(); // Call public method

System.out.println("Age: " + obj2.getAge2()); // Access private variable using getter

}

}

* **CarMain.java** (Day\_5\_Pratic package):

package Day\_5\_Pratic;

public class CarMain {

public String name2 = "Amit Tukaram Kadam"; // Public variable

private int age2 = 292; // Private variable

public void myWork2() {

System.out.println("Going to Work");

}

public int getAge2() { // Getter for private variable

return age2;

}

}

**3. Steps to Compile and Run**

**Step 1: Open Terminal/Command Prompt**

* Navigate to the src folder where your packages are stored.
* Command: cd path/to/your/JavaProject/src

**Step 2: Compile the Java Files**

* Use the javac command to compile the Java files:
* Command:

javac -d . Day\_5\_Pratic/CarMain.java Day\_6\_practice/Car.java Day\_6\_practice/ObjectPractice.java

* This will generate .class files in the corresponding package directories.

**Step 3: Run the Program**

* Run the main class ObjectPractice:
* Command:

java Day\_6\_practice.ObjectPractice

* Expected Output:

Accessing another package class:

Name: Amit Tukaram Kadam

Going to Work

Age: 292

1. **Troubleshooting**

* **Error: Cannot find symbol**
  + Ensure you imported the correct class using import Day\_5\_Pratic.CarMain;
  + Verify the class and package names are correct.
* **Error: Could not find or load main class**
  + Ensure the .class files are compiled in the correct package structure.
  + Run the program from the src directory with the fully qualified class name Day\_6\_practice.ObjectPractice.

**Class vs Object**

### ****1. Class:****

* A **class** is a blueprint or template for creating objects.
* It defines attributes (variables) and behaviors (methods) that the objects will have.
* A class does not occupy memory until an object is created from it.
* Declared using the class keyword in Java.

#### **Example:**

class Car {

String brand;

int speed;

void display() {

System.out.println("Brand: " + brand + ", Speed: " + speed);

}

}

### ****2. Object:****

* An **object** is an instance of a class.
* It represents a real-world entity with state (attributes) and behavior (methods).
* Objects are created using the new keyword.
* Multiple objects can be created from a single class.

#### **Example:**

public class Main {

public static void main(String[] args) {

Car car1 = new Car();

car1.brand = "Toyota";

car1.speed = 120;

car1.display();

}

}

### ****3. Key Differences:****

| **Feature** | **Class** | **Object** |  |
| --- | --- | --- | --- |
| Definition | Blueprint/template | Instance of a class |  |
| Memory | No memory allocation | Takes memory when created |  |
| Example | class Car {} | Car car1 = new Car(); |  |

### ****4. Real-World Analogy:****

* **Class** → Blueprint of a house
* **Object** → Actual house built using the blueprint

### ****5. Important Points:****

* A class can have multiple objects.
* Objects have unique values but share the same structure (class definition).
* Objects interact using methods defined in the class.
* Encapsulation is achieved through classes and objects.
* Constructors initialize objects automatically.

### ****6. Constructor Example:****

class Student {

String name;

int age;

Student(String n, int a) {

name = n;

age = a;

}

void display() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student("John", 20);

s1.display();

}

}

### ****7. Summary:****

* **Class** → Defines structure, does not consume memory until instantiated.
* **Object** → Instance of a class, consumes memory, represents real-world entity.
* **Encapsulation** → Wrapping of data and methods into a class.
* **Constructor** → Special method to initialize objects automatically.

Understanding **classes and objects** is fundamental to mastering **Object-Oriented Programming (OOP).**

**This Keyword in Java**

The this keyword in Java is a reference variable that refers to the **current object** of a class. It helps resolve naming conflicts, invoke methods, call constructors, return the current instance, and pass the current object as an argument.

### ****1. Refers to the Current Instance****

The this keyword represents the current object within a class. It is primarily used inside instance methods or constructors to differentiate between instance variables and parameters.

#### **Example:**

class Example {

int a;

Example(int a) {

this.a = a; // Resolving name conflict

}

}

Here, this.a refers to the instance variable a, while the a on the right-hand side refers to the constructor parameter.

### ****2. Invoking Current Class Methods****

The this keyword can be used to call another method from the same class explicitly.

#### **Example:**

class Test {

void display() {

System.out.println("Display method");

}

void show() {

this.display(); // Calling display() using this

}

public static void main(String[] args) {

Test obj = new Test();

obj.show();

}

}

Here, this.display(); is used to call the display() method from within show().

### ****3. Calling Another Constructor (Constructor Chaining)****

In Java, one constructor can call another constructor in the same class using this(). This technique is known as **constructor chaining**.

#### **Example:**

class Demo {

Demo() {

this(5); // Calls the parameterized constructor

System.out.println("Default Constructor");

}

Demo(int x) {

System.out.println("Parameterized Constructor: " + x);

}

public static void main(String[] args) {

Demo obj = new Demo();

}

}

Here, this(5); calls the parameterized constructor before executing the default constructor.

### ****4. Returning the Current Object****

The this keyword can be used to return the current class instance from a method.

#### **Example:**

class Sample {

Sample getObject() {

return this; // Returning the current instance

}

public static void main(String[] args) {

Sample obj = new Sample();

Sample newObj = obj.getObject();

System.out.println("Object returned: " + newObj);

}

}

The method getObject() returns the current object using this.

### ****5. Passed as an Argument****

The this keyword can be passed as an argument to methods or constructors.

#### **Example: Passing this to a Method**

class Example {

void show(Example obj) {

System.out.println("Method called");

}

void display() {

show(this); // Passing current object

}

public static void main(String[] args) {

Example obj = new Example();

obj.display();

}

}

Here, this is passed to show() to refer to the current object.

#### **Example: Passing this to a Constructor**

class Test {

Test(A obj) {

System.out.println("Constructor called");

}

}

class A {

void create() {

Test t = new Test(this); // Passing current object

}

public static void main(String[] args) {

A obj = new A();

obj.create();

}

}

Here, this is passed to the constructor of Test to reference the current instance of class A.

### ****Key Takeaways****

✅ The this keyword refers to the current instance of a class. ✅ Used to resolve name conflicts between instance variables and parameters. ✅ Helps in calling current class methods and constructors. ✅ Enables constructor chaining within the same class. ✅ Can be returned from a method to refer to the current instance. ✅ Can be passed as an argument to methods and constructors.

By understanding these uses, you can write cleaner and more efficient Java code!

## **static Keyword in Java**

The static keyword in Java is used for memory management. It can be applied to v ariables, methods, blocks, and nested classes.

### Important Points about static:

**Belongs to Class, Not Instance:**

Static members belong to the class itself rather than any specific object.

class Example {

static int count = 0; // Shared among all objects

}

**Used for Memory Efficiency:**

Static members are stored in a common memory area (Class Area) and shared among all instances, reducing memory usage.

**Static Variables:**

Declared with static, they are shared by all instances of the class.

class Counter {

static int count = 0;

Counter() {

count++;

System.out.println("Count: " + count);

}

}

**Static Methods:**

Can be called without creating an object of the class.

class MathUtil {

static int square(int x) {

return x \* x;

}

}

class Test {

public static void main(String[] args) {

System.out.println(MathUtil.square(5)); // No object needed

}

}

**Cannot Use** this **Inside Static Methods:**

Since this refers to an instance and static methods belong to the class, using this inside a static method results in a compilation error.

**Static Blocks (Initialization Block):**

Executed once when the class is loaded into memory.

class Example {

static {

System.out.println("Static Block Executed");

}

}

**Static Nested Class:**

A class inside another class can be declared static.

class Outer {

static class Inner {

void display() {

System.out.println("Static Nested Class");

}

} }

### Key Differences Between this and static:

| **Feature** | **this** | **static** |
| --- | --- | --- |
| Refers To | Current object instance | Class itself |
| Usage | Instance-specific operations | Shared resources across instances |
| Access | Available in instance methods | Available in static methods |
| Memory | Object heap memory | Class memory (Method Area) |
| Method Invocation | Requires object creation | Called without object |

### Summary:

* this - is used to refer to the current instance of a class.
* Static - is used for class-level members that are shared across all instances.
* This - helps in constructor chaining, method invocation, and resolving variable conflicts.
* Static - is useful for memory optimization and accessing class-wide members without creating instances.

## **1. Constructors in Java**

### ****What is a Constructor?****

* A constructor is a special method in Java used to initialize objects.
* It has the same name as the class.
* It does **not** have a return type (not even void).
* It is **automatically invoked** when an object is created.

### ****Types of Constructors****

#### **1.1 Default Constructor (No-Arg Constructor)**

* A constructor without parameters.
* Java provides a **default constructor** if no constructor is explicitly defined.
* Used to assign default values to object attributes.

##### ****Example: Default Constructor****

class Student {

int rollNo;

String name;

// Default Constructor

Student() {

System.out.println("Default Constructor Called");

rollNo = 0;

name = "Unknown";

}

void display() {

System.out.println("Roll No: " + rollNo + ", Name: " + name);

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student(); // Default constructor is invoked

s1.display();

}

}

##### ****Output:****

Default Constructor Called

Roll No: 0, Name: Unknown

#### **1.2 Parameterized Constructor**

* A constructor that takes arguments to initialize an object with specific values.

##### ****Example: Parameterized Constructor****

class Student {

int rollNo;

String name;

// Parameterized Constructor

Student(int r, String n) {

rollNo = r;

name = n;

}

void display() {

System.out.println("Roll No: " + rollNo + ", Name: " + name);

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student(101, "Amol");

s1.display();

}

}

##### ****Output:****

Roll No: 101, Name: Amol

#### **1.3 Copy Constructor**

* Used to create a new object by copying values from an existing object.

##### ****Example: Copy Constructor****

class Student {

int rollNo;

String name;

// Parameterized Constructor

Student(int r, String n) {

rollNo = r;

name = n;

}

// Copy Constructor

Student(Student s) {

rollNo = s.rollNo;

name = s.name;

}

void display() {

System.out.println("Roll No: " + rollNo + ", Name: " + name);

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student(102, "Rahul");

Student s2 = new Student(s1); // Copy constructor invoked

s1.display();

s2.display();

}

}

##### ****Output:****

Roll No: 102, Name: Rahul

Roll No: 102, Name: Rahul

## **2. Code Blocks in Java**

### ****2.1 Static Block****

* A block that runs **only once** when the class is loaded into memory.
* Used to **initialize static variables** or perform setup operations.

##### ****Example: Static Block****

class Demo {

static int a;

// Static Block

static {

System.out.println("Static Block Executed");

a = 10;

}

static void display() {

System.out.println("Value of a: " + a);

}

}

public class Main {

public static void main(String[] args) {

Demo.display();

}

}

##### ****Output:****

Static Block Executed

Value of a: 10

**Static blocks execute before the** main() **method runs.**

### ****2.2 Instance Initialization Block (IIB)****

* Runs **each time an object is created**, before the constructor executes.
* Used to **initialize instance variables**.

##### ****Example: Instance Initialization Block****

class Demo {

int x;

// Instance Initialization Block

{

System.out.println("Instance Initialization Block Executed");

x = 20;

}

Demo() {

System.out.println("Constructor Executed");

}

void display() {

System.out.println("Value of x: " + x);

}

}

public class Main {

public static void main(String[] args) {

Demo d1 = new Demo();

d1.display();

}

}

##### ****Output:****

Instance Initialization Block Executed

Constructor Executed

Value of x: 20

**IIB executes before the constructor when an object is created.**

**Memory Management in Java**

## **1. Introduction to Memory in Java**

Java manages memory automatically using **Garbage Collection (GC)**. The primary types of memory allocation in Java include:

* **Stack Memory**
* **Heap Memory**
* **Method Area (Static Memory)**
* **Metaspace**
* **Direct Memory (Off-Heap Memory)**

## **2. Stack Memory**

Stack memory is used for:

* Method execution and local variables.
* Function calls and thread execution.
* Stores references to heap objects.

### Key Points:

* Each thread has its own stack.
* Faster than heap memory.
* Memory is automatically deallocated after method execution.

### Example:

class StackExample {

void methodA() {

int x = 10; // Stored in stack

methodB();

}

void methodB() {

int y = 20; // Stored in stack

}

public static void main(String[] args) {

StackExample obj = new StackExample();

obj.methodA();

}

}

## 3. Static Memory (Method Area)

Static memory is a part of the **Method Area** in Java, where class-related information is stored. This includes:

* **Class metadata** (structure of a class, methods, and variables)
* **Static variables** (shared across all instances of the class)
* **Static methods** (methods that belong to the class, not an instance)
* **Constant pool** (string literals and final constants)

### Key Points:

* Allocated when the class is loaded into memory.
* Exists until the class is unloaded.
* Common for all instances of a class.
* Memory is not deallocated until JVM shuts down.

### Example:

class Example {

static int count = 0;

static void display() {

System.out.println("Static Method Called");

}

public static void main(String[] args) {

count = 5;

display();

System.out.println("Static Variable: " + count);

}

}

## 4. Heap Memory

Heap memory is used for storing **objects** and **instance variables** at runtime.

* Every object created using new is stored in the heap.
* JVM manages heap memory using **Garbage Collection (GC)**.
* Divided into **Young Generation, Old Generation (Tenured), and Permanent Generation (Metaspace in Java 8+).**

### Key Points:

* Allocated at runtime.
* Objects remain in heap memory until **GC removes them**.
* Memory is shared among all threads.
* Larger compared to stack memory.

### Example:

class Person {

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

void show() {

System.out.println("Name: " + name + ", Age: " + age);

}

public static void main(String[] args) {

Person p1 = new Person("John", 25);

p1.show();

}

}

## 5. Metaspace (Java 8+)

* Replaces **PermGen** (Permanent Generation) in Java 8.
* Stores **class metadata**.
* Dynamically resizable.

### Key Points:

* No fixed size; grows as needed.
* Uses native memory, reducing OutOfMemoryError issues.
* Managed by JVM.

## 6 . Direct Memory (Off-Heap Memory)

* Used for **ByteBuffer operations** (e.g., NIO operations).
* Allocated outside Java heap memory.
* Improves performance in **I/O-intensive applications**.

### Key Points:

* Managed by OS, not JVM.
* Requires explicit deallocation.
* Used in **high-performance applications**.

### Example:

import java.nio.ByteBuffer;

public class DirectMemoryExample {

public static void main(String[] args) {

ByteBuffer buffer = ByteBuffer.allocateDirect(1024);

System.out.println("Direct Buffer Created");

}

}

## 7. Difference Between Memory Types

| **Feature** | **Stack Memory** | **Heap Memory** | **Method Area (Static)** | **Metaspace** | **Direct Memory** |
| --- | --- | --- | --- | --- | --- |
| Storage Type | Local Variables | Objects, Instance Variables | Class Metadata, Static Members | Class Metadata | Byte Buffers |
| Lifetime | Until method ends | Until GC removes | Until class unloads | Managed by JVM | Managed by OS |
| Speed | Fast | Moderate | Moderate | Slow | Fast |
| Management | Automatically deallocated | Managed by GC | Managed by JVM | Managed by JVM | Requires explicit deallocation |

## 8. Garbage Collection in Heap Memory

Java's **Garbage Collector (GC)** automatically removes unused objects from the heap to free memory. Some common GC techniques include:

* **Mark and Sweep** (Identifies and removes unreachable objects)
* **Generational GC** (Divides heap into Young, Old, and Permanent generations)
* **Reference Counting** (Counts references to objects; not used in modern JVMs due to circular reference issues)

## 9. Best Practices for Memory Management

* Use **static variables wisely** to avoid memory leaks.
* Avoid unnecessary object creation; use **reuse patterns (e.g., singleton)**.
* Use StringBuilder instead of String for concatenation to reduce memory usage.
* Close resources (files, database connections) to free heap memory.
* Monitor memory usage using **profilers** (e.g., VisualVM, JConsole).

## 10. Conclusion

Understanding **Java memory management** is crucial for efficient application development. **Stack memory handles method execution, Heap memory stores objects, Static memory stores class-level data, Metaspace handles metadata, and Direct Memory handles off-heap storage.** Proper memory management optimizes performance and prevents memory leaks.

### ****1. Primitive Data Types****

* **Definition:** Stores simple values directly in memory.
* **Size:** Fixed, based on the type.
* **Stored In:** Stack Memory (faster access).

**Examples:**

* **byte** (1 byte) → Range: -128 to 127
* **short** (2 bytes) → Range: -32,768 to 32,767
* **int** (4 bytes) → Range: -2^31 to 2^31-1
* **long** (8 bytes) → Range: -2^63 to 2^63-1
* **float** (4 bytes) → Stores decimal values (less precise)
* **double** (8 bytes) → More precise decimal values
* **char** (2 bytes) → Stores a single Unicode character
* **boolean** (1 bit) → true or false

#### **Characteristics of Primitive Data Types:**

* Stored directly in memory.
* Faster access time.
* Cannot store additional methods or properties.

**Default Values:**

* **int, short, byte, long** → 0
* **float, double** → 0.0
* **char** → '\u0000' (null character)
* **boolean** → false

### ****2. Reference Data Types****

* **Definition:** Stores memory addresses (references) to objects instead of actual data.
* **Size:** Not fixed, depends on the object.
* **Stored In:** Heap Memory (slower access).

**Examples:**

* **String** → "Hello"
* **Arrays** → {1, 2, 3}
* **Objects** (instances of a class)
* **Wrapper Classes** (Integer, Double, etc.)

#### **Characteristics of Reference Data Types:**

* Stores memory address, not actual data.
* Slower than primitive types (due to indirection).
* Default value: **null** (if not initialized).
* Can have methods and properties.

### ****3. Key Differences Between Primitive and Reference Types****

| **Feature** | **Primitive Type** | **Reference Type** |
| --- | --- | --- |
| **Storage** | Stored in Stack Memory | Stored in Heap Memory |
| **Contains** | Actual value | Memory address (reference) |
| **Speed** | Faster | Slower (due to indirect memory access) |
| **Size** | Fixed | Varies (depends on object) |
| **Default Value** | 0, false, \u0000 | null |
| **Examples** | int, double, char | String, Array, Object |

### ****4. Important Notes****

#### **Pass by Value:**

* **Primitive types** pass **copies of values** to methods.
* **Reference types** pass **memory addresses** (references). Changing the object inside a method affects the original object outside the method.

#### **Wrapper Classes (Integer, Double, etc.):**

* Wrapper classes are used to convert primitive types into objects.

**Example:**

int x = 5; Integer y = x; // Auto-boxing (primitive to reference)int z = y; // Unboxing (reference to primitive)

#### **String is Immutable:**

* Although **String** is a reference type, it behaves like a primitive because it **cannot be changed** after creation.

**Example:**

String s1 = "Hello";// String s1 cannot be modified directly. Any changes would result in a new String object.

# **Variable Scope in Java**

In Java, **variable scope** refers to the region of a program where a variable can be accessed. It defines where a variable is visible and usable within the program. The scope of a variable is determined by where it is declared in the code. Java has several types of variable scopes, which are as follows:

1. **Local Variable Scope**
2. **Instance Variable Scope**
3. **Class (Static) Variable Scope**
4. **Variable Shadowing**
5. **Block-Level Variable Scope**
6. **Global Variables** (Note: Java doesn't have true global variables)

## **1. Local Variable Scope**

* **Definition**: Local variables are declared inside methods, constructors, or blocks. Their scope is limited to the block or method in which they are declared.
* **Access**: Local variables can only be accessed within the method, constructor, or block in which they are declared.
* **Lifetime**: Local variables are created when the method is called and destroyed when the method finishes execution.

### ****Example****:

public class LocalVariableExample {

public void exampleMethod() {

int localVar = 5; // Local variable

System.out.println(localVar); // Can be accessed here

}

public void anotherMethod() {

// System.out.println(localVar); // Error! Cannot access localVar here

}

}

In this example, localVar is only accessible inside exampleMethod(). Trying to access it outside this method will result in an error.

## **2. Instance Variable Scope**

* **Definition**: Instance variables are declared inside a class but outside any method. They represent the state of an object and are accessible throughout the class.
* **Access**: Instance variables can be accessed from any method or constructor within the class, provided you have an object of that class.
* **Lifetime**: Instance variables are created when an object is instantiated and destroyed when the object is garbage collected.

### ****Example****:

public class InstanceVariableExample {

int instanceVar = 10; // Instance variable

public void displayVar() {

System.out.println(instanceVar); // Can be accessed in any method of the class

}

public void modifyVar() {

instanceVar = 20; // Can modify the instance variable

}

}

In this example, instanceVar is accessible in all methods of the InstanceVariableExample class.

## **3. Class (Static) Variable Scope**

* **Definition**: Class variables (or static variables) are declared with the static keyword. These variables are shared among all instances of the class.
* **Access**: Class variables can be accessed from any static or non-static method of the class.
* **Lifetime**: Class variables are created when the class is loaded and destroyed when the class is unloaded.

### ****Example****:

public class StaticVariableExample {

static int staticVar = 30; // Static variable

public static void displayVar() {

System.out.println(staticVar); // Can be accessed directly in a static method

}

public void modifyVar() {

staticVar = 40; // Can be accessed and modified in a non-static method

}

}

In this example, staticVar is a class variable. It is shared across all instances of the StaticVariableExample class.

## **4. Variable Shadowing**

* **Definition**: Shadowing occurs when a local variable has the same name as an instance or class variable. The local variable "shadows" the instance or class variable inside the method or block.
* **Access**: The local variable takes precedence inside the method or block, hiding the instance or class variable.

### ****Example****:

public class ShadowingExample {

int x = 10; // Instance variable

public void myMethod() {

int x = 20; // Local variable that shadows the instance variable

System.out.println(x); // Prints the local variable x (20), not the instance variable

}

}

In this example, the local variable x inside myMethod() shadows the instance variable x, and the local x is used within the method.

## **5. Block-Level Variable Scope**

* **Definition**: Variables can be declared inside specific code blocks like loops, conditionals, or other block statements. These variables are accessible only within the block where they are defined.
* **Lifetime**: Block-level variables are created when the block is entered and destroyed when the block is exited.

### ****Example****:

public class BlockVariableExample {

public void exampleMethod() {

if (true) {

int blockVar = 50; // Block-level variable

System.out.println(blockVar); // Can be accessed here inside the block

}

// System.out.println(blockVar); // Error! Cannot access blockVar here

}

}

In this example, blockVar is only accessible within the if block and cannot be accessed outside the block.

**6. Global Variables (Not Applicable in Java)**

* **Note**: Java does not support true global variables, which are accessible throughout the entire program. However, static variables in a class can behave similarly to global variables since they are shared across all instances of the class.

## **Summary Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable Type** | **Scope** | **Lifetime** | **Example** |
| **Local Variable** | Inside a method/block | Exists only during the method/block execution | int x = 5; inside myMethod() |
| **Instance Variable** | Entire class (through instance) | Exists as long as the object exists | int x = 10; in the class, can be accessed by all methods |
| **Class Variable (Static)** | Entire class (shared across instances) | Exists as long as the class is loaded | static int x = 30; in the class, accessed by all instances |
| **Shadowing Variable** | Within method/block where shadowed | Lifetime depends on the method/block | int x = 5; inside method shadows class variable x |
| **Block-Level Variable** | Inside the block (loop, conditional) | Exists only within the block | int x = 10; inside if block |

**Java Garbage Collection**

**🧹 What is Garbage Collection?**

* In Java, **Garbage Collection (GC)** means removing unused (unreachable) objects from memory.
* It is automatic and handled by the **JVM**.
* It helps keep memory clean and prevents memory leaks.

**🧠 Why Garbage Collection is Needed?**

* Java programs create many objects. Not all are needed forever.
* GC removes unnecessary objects to **free memory**.
* Prevents errors like OutOfMemoryError.

**🔍 When is an Object Eligible for GC?**

An object becomes eligible when **no live reference** is pointing to it.

**Simple Examples:**

1. **Setting reference to null**

Student s = new Student();

s = null; // Now object is unreachable

1. **Reassigning reference**

Student s1 = new Student();

Student s2 = new Student();

s1 = s2; // Old s1 object has no reference now

1. **Local variable in a method**

void display() {

Student s = new Student();

} // After method ends, 's' is gone

1. **Island of Isolation**

A a = new A();

B b = new B();

a.b = b;

b.a = a;

a = null;

b = null; // Both become unreachable

**🔄 Steps in Garbage Collection**

**✅ 1. Object Creation**

Objects are made in heap memory.

**✅ 2. Reference Checking**

JVM checks which objects are still connected (reachable).

**✅ 3. Mark and Sweep**

* **Mark**: Mark all reachable objects.
* **Sweep**: Delete unmarked (unreachable) objects.

**✅ 4. Finalize (Deprecated)**

If finalize() is present, JVM may call it before deleting the object.

protected void finalize() {

System.out.println("Object is being deleted");

}

But this is old and should be avoided now.

**🌐 Types of Garbage Collectors in Java**

|  |  |  |
| --- | --- | --- |
| **GC Type** | **JVM Option** | **Used For** |
| **Serial GC** | -XX:+UseSerialGC | Simple applications |
| **Parallel GC** | -XX:+UseParallelGC | High-performance apps |
| **CMS (Concurrent Mark Sweep)** | -XX:+UseConcMarkSweepGC | Low pause time apps |
| **G1 GC (Garbage First)** | -XX:+UseG1GC | Large apps, default now |
| **ZGC** | -XX:+UseZGC | Low latency, large memory (Java 11+) |
| **Shenandoah** | -XX:+UseShenandoahGC | Very low pause (Java 12+) |

**⚡ How to Suggest Garbage Collection**

System.gc();

Runtime.getRuntime().gc();

* These suggest the JVM to run GC, but JVM may choose to ignore.

**❌ About finalize() Method**

* Used in old Java to clean up before deleting object.
* Now **deprecated** (Java 9+).
* Use try-with-resources or close() instead.

**📅 Best Practices for Good Memory Use**

1. Use **local variables** when possible.
2. Don’t keep unnecessary **static references**.
3. Manually set unused objects to **null**.
4. Use **try-with-resources** to auto-close files.
5. Use tools like VisualVM or MAT to monitor memory.

**📝 Summary**

* GC in Java happens automatically.
* It removes unreachable objects from memory.
* Helps improve performance and avoid memory issues.
* Developers don’t control GC directly, but can write memory-efficient code.

**⚖ Tools to Monitor Garbage Collection**

* **VisualVM**
* **JConsole**
* **Java Mission Control**
* **Eclipse MAT**
* Enable **GC Logs** using JVM flags

📘 **finalize() Method in Java**

🔹 **Overview**

* The finalize() method is a special method in Java used for cleanup operations before an object is destroyed.
* Purpose: Perform cleanup before the object is garbage collected.
* Defined in: java.lang.Object

🔹 **Method Signature**

protected void finalize() throws Throwable

* protected: Accessible within the same package or subclasses.
* void: Does not return any value.
* throws Throwable: Can throw any object of type Throwable.

🔹 **Purpose of finalize()**

* Acts similarly to a destructor in other languages like C++.
* Common use cases include:
  + Closing file streams
  + Releasing network connections
  + Cleaning up memory-intensive resources

🔹 **How finalize() Works**

1. An object becomes unreachable.
2. The Garbage Collector (GC) detects it.
3. If the finalize() method is overridden, it is called once.
4. The object may be resurrected during this call.
5. If not resurrected, the object is garbage collected.

🔹 **Code Example**

class MyClass {

@Override

protected void finalize() throws Throwable {

try {

System.out.println("Finalize called on: " + this);

} finally {

super.finalize();

}

}

}

public class FinalizeExample {

public static void main(String[] args) {

MyClass obj = new MyClass();

obj = null;

System.gc();

System.out.println("Main ends");

}

}

**Expected Output:**

Main ends

Finalize called on: MyClass@XYZ

🔹 **Important Points**

* ❗ Not guaranteed to be called immediately after object becomes unreachable.
* 🔁 Called only once per object.
* ❌ Deprecated since Java 9.
* 🧼 Should not be used for critical cleanup tasks.
* 🐢 Can delay garbage collection.

🔹 **Alternatives to finalize()**

* try-with-resources (implements AutoCloseable)
* Cleaner API (in java.lang.ref.Cleaner)

🔹 **Deprecated Warning**

@Deprecated(since = "9")

protected void finalize() throws Throwable

**Reasons for deprecation:**

* Unpredictable execution timing
* Performance degradation
* Availability of better alternatives

🔹 **Best Practices**

* ✅ Avoid using finalize()
* ✅ Use try-with-resources for managing I/O and other resources
* ✅ Use Cleaner for background cleanup
* ✅ Rely on explicit resource management whenever possible

🔹 **Common Interview Questions**

* What is the use of finalize()?
* When is it called?
* Is finalize() guaranteed to execute?
* Why is finalize() deprecated?
* What are the alternatives?

🔚 **Conclusion**

* The finalize() method was meant for cleanup before object destruction.
* It is deprecated due to unreliability and better alternatives.
* Always prefer modern, explicit, and predictable resource management techniques like try-with-resources and Cleaner.

**Ternary Operator in Java**

**What is the Ternary Operator?**

The **Ternary Operator** is a shorthand version of the **if-else** statement. It is used to evaluate boolean expressions and return one of two values depending on the result.

**Syntax:**

condition ? expression1 : expression2;

* If condition is **true**, the result is expression1
* If condition is **false**, the result is expression2

**Example 1: Basic Usage**

int a = 10, b = 20;

int max = (a > b) ? a : b;

System.out.println("Maximum: " + max); // Output: 20

**Example 2: String Output**

int age = 18;

String result = (age >= 18) ? "Eligible" : "Not Eligible";

System.out.println(result); // Output: Eligible

**Example 3: Nested Ternary Operator**

int num = 0;

String type = (num > 0) ? "Positive" : (num < 0) ? "Negative" : "Zero";

System.out.println(type); // Output: Zero

Note: Use nested ternary operators carefully to maintain code readability.

**Ternary vs If-Else**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Ternary Operator** | **if-else Statement** |
| Syntax | Compact (one-liner) | Multi-line |
| Readability | Less readable if complex | More readable |
| Use Case | Simple conditions | Complex logic |

**Ternary Operator with Functions**

public static String checkEvenOdd(int num) {

return (num % 2 == 0) ? "Even" : "Odd";

}

**Ternary Operator in Assignment**

int x = 5;

int y = 10;

int result = (x > y) ? x + 10 : y + 10;

System.out.println(result); // Output: 20

**Type Compatibility in Ternary Operator**

Both expressions should return compatible types.

int x = 10;

double y = 20.5;

double result = (x > 5) ? x : y; // Valid, promotes int to double

Invalid example:

String str = (x > 5) ? "Yes" : 10; // Error – incompatible types

**When to Avoid Ternary Operator**

Avoid using ternary operator for:

* Multiple statements
* Deeply nested conditions
* Complex logic  
  Use if-else instead for better clarity.

**Summary Points**

* Shorthand for if-else
* Compact and clean for simple decisions
* Syntax: condition ? expr1 : expr2;
* Nesting possible but not recommended for complex logic
* Useful in assignments and return statements

**Java Switch Statement**

**1. Introduction to Switch Statement**

The switch statement in Java is a control flow statement that allows variable values to be tested for equality against a list of values, each called a case. It is often used as a more readable alternative to long chains of if-else-if statements.

**2. Syntax of Switch Statement**

switch (expression) {

case value1:

// code block

break;

case value2:

// code block

break;

// more cases...

default:

// default code block

}

**Components:**

* expression: Must evaluate to byte, short, char, int, String, or enum.
* case: Each value must be a constant or literal.
* break: Optional, but usually used to terminate a case.
* default: Optional, executes if no case matches.

**3. Rules for Using Switch Statement**

* No duplicate case values are allowed.
* The switch expression must result in a valid data type (byte, short, char, int, enum, String, etc.).
* The break statement terminates the current case.
* The default case is optional and can be placed anywhere.

**4. Example: Basic Switch Statement**

int day = 3;

switch (day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

case 3:

System.out.println("Wednesday");

break;

default:

System.out.println("Invalid day");

}

**Output:**

Wednesday

**5. Example: Switch with String**

String fruit = "Apple";

switch (fruit) {

case "Apple":

System.out.println("Fruit is Apple");

break;

case "Mango":

System.out.println("Fruit is Mango");

break;

default:

System.out.println("Unknown fruit");

}

**6. Example: Switch without Break (Fall-through)**

int number = 2;

switch (number) {

case 1:

System.out.println("One");

case 2:

System.out.println("Two");

case 3:

System.out.println("Three");

default:

System.out.println("Default case");

}

**Output:**

Two

Three

Default case

**7. Nested Switch Statement**

int num = 1;

char ch = 'A';

switch (num) {

case 1:

switch (ch) {

case 'A':

System.out.println("Nested A");

break;

default:

System.out.println("Nested default");

}

break;

default:

System.out.println("Outer default");

}

**8. Switch Expression (Java 14+)**

Java 14 introduced switch as an expression.

int day = 3;

String result = switch (day) {

case 1 -> "Monday";

case 2 -> "Tuesday";

case 3 -> "Wednesday";

default -> "Invalid day";

};

System.out.println(result);

**9. Difference between if-else and switch**

|  |  |  |
| --- | --- | --- |
| **Feature** | **if-else** | **switch** |
| Expression type | Any boolean condition | Specific values (int, char, String) |
| Flexibility | More flexible | Limited to certain data types |
| Performance | Slower for many conditions | Faster with many discrete values |
| Readability | Less readable with many cases | Cleaner and more readable |
| Suitable for | Complex conditions | Fixed value checks |

**10. When to Use What?**

* Use switch when checking a single variable against a list of constant values.
* Use if-else when checking complex conditions, ranges, or expressions.

**11. Conclusion**

The switch statement in Java is a powerful tool for simplifying the selection of code blocks based on constant values. While if-else is more flexible, switch is often more readable and concise when dealing with known, discrete values.

**Java Loops**

**What is a Loop in Java?**

Loops in Java are used to execute a block of code repeatedly under certain conditions.

**Types of Loops in Java**

**1. For Loop**

Used when the number of iterations is known.

for (initialization; condition; update) {

// code block

}

**Example:**

for (int i = 1; i <= 5; i++) {

System.out.println(i);

}

**Flow:**

1. Initialization
2. Condition check
3. Execute block
4. Update
5. Repeat

**2. While Loop**

Used when the number of iterations is not known in advance.

while (condition) {

// code block

}

**Example:**

int i = 1;

while (i <= 5) {

System.out.println(i);

i++;

}

**Flow:**

1. Check condition
2. Execute block
3. Update
4. Repeat

**3. Do-While Loop**

Same as while loop, but it executes at least once even if the condition is false.

do {

// code block

} while (condition);

**Example:**

int i = 1;

do {

System.out.println(i);

i++;

} while (i <= 5);

**Enhanced For Loop (For-Each Loop)**

Used to iterate through elements in arrays or collections.

for (dataType item : array) {

// code block

}

**Example:**

int[] numbers = {1, 2, 3, 4, 5};

for (int num : numbers) {

System.out.println(num);

}

**Loop Control Statements**

**break Statement**

Used to exit the loop prematurely.

for (int i = 1; i <= 10; i++) {

if (i == 5) break;

System.out.println(i);

}

**continue Statement**

Skips the current iteration and jumps to the next one.

for (int i = 1; i <= 5; i++) {

if (i == 3) continue;

System.out.println(i);

}

**Nested Loops**

Using a loop inside another loop.

for (int i = 1; i <= 3; i++) {

for (int j = 1; j <= 2; j++) {

System.out.println("i=" + i + ", j=" + j);

}

}

**Labeled Loops**

Useful when breaking out of nested loops.

outer:

for (int i = 1; i <= 3; i++) {

for (int j = 1; j <= 3; j++) {

if (j == 2) break outer;

System.out.println(i + " " + j);

}

}

**When to Use Which Loop?**

|  |  |
| --- | --- |
| **Use Case** | **Loop Type** |
| Known number of iterations | for |
| Unknown iterations | while |
| At least once execution | do-while |
| Traversing arrays/lists | for-each |

**Difference Between All Loops**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Feature** | **for Loop** | **while Loop** | **do-while Loop** | **for-each Loop** |
| Initialization | Inside loop header | Before loop | Before loop | Not applicable |
| Condition Check | At the beginning | At the beginning | After executing the loop | Automatically handled |
| Execution Guarantee | 0 or more times | 0 or more times | At least once | 0 or more (based on array) |
| Use Case | Known iterations | Unknown iterations | Execute once, then check | Array or collection |
| Control | Full control | Full control | Full control | Limited (no index access) |

**Best Practices**

* Avoid infinite loops unless required.
* Always update loop variables.
* Prefer for-each for simple collection iteration.
* Use meaningful labels if using labeled loops.

**Recursion in Programming**

**1. Self-Calling Function**  
Recursion occurs when a function calls itself. This allows a function to repeat its behavior until a base case is met.

**Example:**

public class Factorial {

public static int factorial(int n) {

if (n == 0) { // Base Case

return 1;

} else { // Recursive Case

return n \* factorial(n - 1);

}

}

public static void main(String[] args) {

System.out.println(factorial(5)); // Output: 120

}

}

**Explanation:**  
In the example above, the factorial function calls itself, reducing n by 1 until it reaches the base case (n == 0).

**2. Base Case**  
The base case is the condition that stops the recursion, preventing infinite loops and allowing the function to return a result.

**Example:**

public class Fibonacci {

public static int fibonacci(int n) {

if (n <= 1) { // Base Case

return n;

} else {

return fibonacci(n - 1) + fibonacci(n - 2); // Recursive Case

}

}

public static void main(String[] args) {

System.out.println(fibonacci(6)); // Output: 8

}

}

**Explanation:**  
The base case in the fibonacci function is when n is less than or equal to 1. This stops the recursion.

**3. Recursive Case**  
The recursive case is where the function calls itself to break the problem into smaller subproblems.

**Example:**

public class Sum {

public static int sum(int n) {

if (n == 0) { // Base Case

return 0;

} else { // Recursive Case

return n + sum(n - 1);

}

}

public static void main(String[] args) {

System.out.println(sum(5)); // Output: 15

}

}

**Explanation:**  
In this example, the function sums numbers from n to 0 by calling itself with n - 1 until the base case is reached.

**4. Stack Overflow Risk**  
Excessive recursion can lead to stack overflow errors, especially when there are too many recursive calls without reaching the base case. This happens because each function call consumes stack space.

**Example:**

public class StackOverflowExample {

public static void infiniteRecursion() {

infiniteRecursion(); // No Base Case

}

public static void main(String[] args) {

infiniteRecursion(); // Stack Overflow

}

}

**Explanation:**  
This code will result in a stack overflow because the recursive function infiniteRecursion() does not have a base case, causing it to call itself indefinitely.

**5. Problem Solving with Recursion**  
Recursion is ideal for problems that can be divided into smaller, similar subproblems. Common examples include factorial calculations, Fibonacci numbers, tree traversals, and searching/sorting algorithms.

**Example (Binary Search):**

public class BinarySearch {

public static int binarySearch(int[] arr, int target, int low, int high) {

if (low > high) { // Base Case: Target not found

return -1;

}

int mid = (low + high) / 2;

if (arr[mid] == target) { // Base Case: Target found

return mid;

} else if (arr[mid] > target) {

return binarySearch(arr, target, low, mid - 1); // Recursive Case: Search left

} else {

return binarySearch(arr, target, mid + 1, high); // Recursive Case: Search right

}

}

public static void main(String[] args) {

int[] arr = {1, 3, 5, 7, 9, 11};

System.out.println(binarySearch(arr, 5, 0, arr.length - 1)); // Output: 2

} }

**Explanation:**  
The binary search algorithm is a great example of recursion, where the problem (finding the target) is divided into smaller subproblems, either to the left or right half of the array.

**Key Points to Remember:**

* **Self-Calling Function**: The function calls itself to solve smaller instances of the same problem.
* **Base Case**: The condition that terminates the recursion.
* **Recursive Case**: The part of the function where recursion occurs.
* **Stack Overflow Risk**: Recursion without proper base cases can lead to a stack overflow error.
* **Problem Solving**: Recursion is helpful for problems that can be divided into smaller, similar subproblems.

**Random Numbers & Math Class**

**1. Math Class Overview**

* **Static Class**:  
  The Math class is **final** and all its methods are **static**, so you can use them without creating an object.
* Math.methodName(arguments);
* **Constants Available**:

| **Constant** | **Description** |
| --- | --- |
| Math.PI | Value of π (3.14159...) |
| Math.E | Base of natural logarithms (2.718...) |

**🧮 2. Key Methods of Math Class**

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Example** |
| abs(x) | Returns absolute value of x | Math.abs(-5) → 5 |
| ceil(x) | Rounds x **up** to nearest integer | Math.ceil(3.2) → 4.0 |
| floor(x) | Rounds x **down** to nearest integer | Math.floor(3.7) → 3.0 |
| round(x) | Rounds x to **nearest** integer | Math.round(2.6) → 3 |
| max(x, y) | Returns **maximum** of x and y | Math.max(10, 20) → 20 |
| min(x, y) | Returns **minimum** of x and y | Math.min(10, 20) → 10 |
| pow(x, y) | Calculates x raised to power y | Math.pow(2, 3) → 8.0 |
| sqrt(x) | Returns square root of x | Math.sqrt(25) → 5.0 |
| cbrt(x) | Returns cube root of x | Math.cbrt(27) → 3.0 |
| exp(x) | Calculates e^x | Math.exp(1) → 2.718... |
| log(x) | Natural log (base e) of x | Math.log(Math.E) → 1.0 |
| log10(x) | Base-10 logarithm | Math.log10(100) → 2.0 |
| toDegrees(x) | Converts radians to degrees | Math.toDegrees(π) → 180.0 |
| toRadians(x) | Converts degrees to radians | Math.toRadians(180) → π |
| signum(x) | Returns sign of a number | Math.signum(-10.5) → -1.0 |
| copySign(m, s) | Returns first argument with the sign of the second | Math.copySign(4.5, -1) → -4.5 |
| hypot(x, y) | Returns √x² + y² (Euclidean distance) | Math.hypot(3, 4) → 5.0 |
| IEEEremainder(x, y) | Remainder of x/y based on IEEE 754 | Math.IEEEremainder(10, 3) → 1.0 |

**3. Generating Random Numbers**

**A. Math.random()**

* **Syntax**:
* double randomValue = Math.random();
* **Range**: Returns a random double between **0.0 (inclusive)** and **1.0 (exclusive)**
* **Random Integer in a Range**:
* int randInt = (int)(Math.random() \* (max - min + 1)) + min;

Example:

int dice = (int)(Math.random() \* 6) + 1; // 1 to 6

**B. java.util.Random Class**

* **Import Required**:
* import java.util.Random;
* **Creating Object**:
* Random rand = new Random();
* **Common Methods**:

|  |  |
| --- | --- |
| **Method** | **Description** |
| nextInt() | Returns random int value |
| nextInt(bound) | Returns random int between 0 (inclusive) and bound (exclusive) |
| nextDouble() | Returns random double between 0.0 and 1.0 |
| nextBoolean() | Returns true or false randomly |
| nextFloat() | Returns random float between 0.0 and 1.0 |
| nextLong() | Returns random long value |

**C. java.security.SecureRandom (For Cryptographic Purposes)**

* **Import Required**:
* import java.security.SecureRandom;
* **Creating Object**:
* SecureRandom secureRand = new SecureRandom();
* **Use Case**: Better randomness than Random, used in security-sensitive applications.
* **toString() Method in Java**

**ToString Method**

1. **Function**: toString() provides a string representation of an object.
2. **Inheritance**: It's inherited from the Object class.
3. **Default Format**: By default, returns class name, "@", and hashcode.
4. **Overriding**: Commonly overridden in custom classes for meaningful output.

**1. Java: toString() Method**

**What is toString()?**

* toString() is a method defined in the Object class in Java.
* It is used to return a string representation of an object.
* When you print an object, the toString() method is automatically called.

**Syntax:**

public String toString()

**Default Behavior:**

* If not overridden, it returns: ClassName@HashCode

**Example:**

class Student {

int id = 101;

String name = "Amol";

public String toString() {

return "Student ID: " + id + ", Name: " + name;

}

public static void main(String[] args) {

Student s1 = new Student();

System.out.println(s1); // toString() is called

}

}

**Output:**

Student ID: 101, Name: Amol

**Why Override toString()?**

* To provide meaningful output when printing objects.
* Helpful in debugging and logging.

**2. JavaScript: toString() Method**

**What is toString()?**

* In JavaScript, toString() is a built-in method available for many objects.
* It converts an object to a string.

**Syntax:**

object.toString()

**Works with:**

* Arrays
* Numbers
* Strings
* Dates
* Custom objects

**Examples:**

**Numbers:**

let num = 123;

console.log(num.toString()); // "123"

**Arrays:**

let arr = [1, 2, 3];

console.log(arr.toString()); // "1,2,3"

**Dates:**

let date = new Date();

console.log(date.toString()); // e.g., "Fri May 02 2025 10:20:00 GMT+0530 (India Standard Time)"

**Custom Objects:**

let person = {

name: "Amol",

age: 25,

toString: function() {

return this.name + " is " + this.age + " years old.";

}

};

console.log(person.toString()); // "Amol is 25 years old."

**Important Points:**

* Overriding toString() in custom objects helps display meaningful string output.
* Automatically called when concatenating with a string or printing.

**Summary:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Language** | **Purpose** | **Can be Overridden** | **Default Behavior** |
| Java | Represents object as a string | Yes | ClassName@Hashcode |
| JavaScript | Converts object to string | Yes (in objects) | Varies by object (array, date, etc.) |

**Use Cases:**

* Debugging
* Logging
* Displaying readable output
* Custom formatting

**Java String Class**

**1. Immutability**

* Once created, a String object's value **cannot be changed**.
* Any modification results in the creation of a **new String object**.
* This ensures **security, thread-safety**, and allows for **caching optimizations**.

String s = "Hello";

s = s.concat(" World"); // "Hello World" is a new object

**2. String Pool (Interning)**

* Java maintains a special memory area called the **String Constant Pool**.
* When a string literal is created:
  + If it already exists in the pool, the existing object is reused.
  + If not, it is added to the pool.
* Using new String() bypasses the pool and creates a new object in the heap.

String a = "Java";

String b = "Java";

String c = new String("Java");

System.out.println(a == b); // true

System.out.println(a == c); // false

System.out.println(a.equals(c)); // true

**3. Comparing Strings**

* == compares references (memory addresses).
* .equals() compares values (content).
* .compareTo() compares lexicographically.

System.out.println("apple".compareTo("banana")); // -1

System.out.println("hello".equals("hello")); // true

System.out.println("hello" == "hello"); // true (due to string pool)

**4. Concatenation**

* Strings can be concatenated using the **+ operator** or the **concat()** method.
* Each concatenation creates a **new String object** due to immutability.

String s1 = "Java" + "Script";

String s2 = "Java".concat("Script");

* For **multiple or dynamic operations**, prefer **StringBuilder** or **StringBuffer** for better performance.

**5. String Methods (Commonly Used)**

|  |  |
| --- | --- |
| **Method** | **Description** |
| length() | Returns the length of the string |
| charAt(index) | Returns the character at the index |
| substring(start,end) | Returns substring from start to end-1 |
| indexOf(char) | Returns index of first occurrence |
| toLowerCase() | Converts string to lowercase |
| toUpperCase() | Converts string to uppercase |
| trim() | Removes leading and trailing whitespace |
| replace(old, new) | Replaces characters or substrings |
| equalsIgnoreCase() | Compares strings ignoring case |
| startsWith(prefix) | Checks if string starts with given prefix |
| endsWith(suffix) | Checks if string ends with given suffix |

**6. Memory and Performance**

* Being **immutable**, each modification creates a new object.
* Can lead to **higher memory usage** during frequent modifications.
* **StringBuilder** and **StringBuffer** are mutable and preferred for repetitive modifications.

**7. String Formatting (Print Format)**

* Use String.format() for formatting:

String formatted = String.format("Name: %s, Age: %d", "John", 25);

System.out.println(formatted); // Output: Name: John, Age: 25

**Format Specifiers**

|  |  |
| --- | --- |
| **Specifier** | **Description** |
| %s | String |
| %d | Integer (decimal) |
| %f | Floating-point (decimal) |
| %c | Character |
| %b | Boolean |
| %n | Newline |

**Width, Precision, Alignment**

System.out.printf("%-10s | %5d | %.2f%n", "Item", 10, 12.345);

// Output: Item | 10 | 12.35

* %-10s: Left-align string in a 10-character field
* %5d: Right-align integer in a 5-character field
* %.2f: Floating-point with 2 decimal places

**8. String Interning**

* You can manually add a string to the pool using .intern()

String s1 = new String("Hello");

String s2 = s1.intern(); // refers to object in string pool

**9. Splitting and Joining**

* Use split() to divide a string using regex.
* Use String.join() to combine strings.

String data = "a,b,c";

String[] parts = data.split(",");

String result = String.join("-", parts); // "a-b-c"

**10. Converting Between String and Other Types**

// String to int

int num = Integer.parseInt("123");

// int to String

String str = String.valueOf(123);

**StringBuffer vs StringBuilder in Java**

**1. Introduction**

In Java, both StringBuffer and StringBuilder are classes used to handle mutable sequences of characters. The primary difference between them is related to synchronization. Let's understand both classes and how they are used in Java programming.

**2. StringBuffer**

**Definition**

StringBuffer is a class in Java that allows us to create objects of strings that can be modified after they are created. It is primarily used when we need to perform multiple modifications to strings, such as appending, inserting, or deleting characters.

**Thread Safety**

The operations of StringBuffer are **synchronized**, meaning it is thread-safe. This means that only one thread can access the method at a time. This makes StringBuffer slower in performance compared to StringBuilder.

**Methods**

* append(String str) – Appends a string to the current StringBuffer object.
* insert(int offset, String str) – Inserts a string at the specified index.
* delete(int start, int end) – Deletes a portion of the string.
* reverse() – Reverses the content of the StringBuffer.

**Example**

public class StringBufferExample {

public static void main(String[] args) {

// Creating a StringBuffer object

StringBuffer buffer = new StringBuffer("Hello");

// Append operation

buffer.append(" World");

System.out.println(buffer); // Output: Hello World

// Insert operation

buffer.insert(5, ",");

System.out.println(buffer); // Output: Hello, World

// Reverse operation

buffer.reverse();

System.out.println(buffer); // Output: dlroW ,olleH

}

}

**3. StringBuilder**

**Definition**

StringBuilder is similar to StringBuffer but with **no synchronization**. It is used to perform string modifications in situations where thread safety is not a concern. It is generally faster than StringBuffer due to the lack of synchronization.

**Thread Safety**

Unlike StringBuffer, StringBuilder does **not** guarantee thread safety. This means that multiple threads can modify a StringBuilder object at the same time, which could cause unexpected results unless proper synchronization is implemented externally.

**Methods**

* append(String str)
* insert(int offset, String str)
* delete(int start, int end)
* reverse()

**Example**

public class StringBuilderExample {

public static void main(String[] args) {

// Creating a StringBuilder object

StringBuilder builder = new StringBuilder("Hello");

// Append operation

builder.append(" World");

System.out.println(builder); // Output: Hello World

// Insert operation

builder.insert(5, ",");

System.out.println(builder); // Output: Hello, World

// Reverse operation

builder.reverse();

System.out.println(builder); // Output: dlroW ,olleH

}

}

**4. Key Differences Between StringBuffer and StringBuilder**

|  |  |  |
| --- | --- | --- |
| **Feature** | **StringBuffer** | **StringBuilder** |
| Thread Safety | Thread-safe (synchronized methods) | Not thread-safe |
| Performance | Slower due to synchronization | Faster, no synchronization |
| Use Case | Multi-threaded environments | Single-threaded environments |
| Methods | Similar to StringBuilder | Similar to StringBuffer |
| Introduced in | Java 1.0 | Java 5 (JDK 1.5) |

**5. Performance Comparison**

StringBuffer can be slower than StringBuilder because of the overhead of synchronization, which is necessary for thread safety. In single-threaded applications, StringBuilder should be preferred as it is faster.

**6. When to Use Which?**

* **Use StringBuffer when:**
  + You need a mutable string and thread safety is important.
  + The code runs in a multi-threaded environment, and multiple threads modify the same string.
* **Use StringBuilder when:**
  + You need a mutable string, but thread safety is not required.
  + The code runs in a single-threaded environment, and performance is critical.

**Java final Keyword**

The final keyword in Java is a modifier used with variables, methods, and classes. It helps to restrict the user in various ways and is used for security and immutability.

**1. Final Variable**

**Definition:**  
A variable declared with final cannot be reassigned once it has been assigned a value.

**Syntax:**

final int x = 10;

x = 20; // Error: cannot assign a value to final variable x

**Use Case:**

* For constants (e.g., final double PI = 3.14159;)

**Note:**

* For final reference variables (objects), the reference cannot be changed, but the object's internal state can change.

final Student s = new Student();

s.name = "Amol"; // Allowed

s = new Student(); // Error

**2. Final Method**

**Definition:**  
A method declared as final cannot be overridden by subclasses.

**Syntax:**

class A {

final void show() {

System.out.println("Hello");

}

}

class B extends A {

void show() { // Error: Cannot override the final method from A

System.out.println("Hi");

}

}

**Use Case:**

* To prevent modification of a method’s behavior in subclasses.

**3. Final Class**

**Definition:**  
A class declared as final cannot be extended (inherited).

**Syntax:**

final class Vehicle {

void run() {

System.out.println("Vehicle is running");

}

}

class Bike extends Vehicle { // Error: Cannot inherit from final class

void run() {

System.out.println("Bike is running");

}

}

**Use Case:**

* To prevent inheritance for security or design reasons.

**4. Final Parameters**

**Definition:**  
A method parameter declared as final cannot be changed inside the method.

**Syntax:**

void display(final int x) {

x = 10; // Error: cannot assign a value to final parameter x

System.out.println(x);

}

**Summary Table:**

|  |  |  |
| --- | --- | --- |
| **Usage** | **Purpose** | **Can be Modified?** |
| Final Variable | Value cannot be reassigned | No |
| Final Method | Method cannot be overridden | No (in subclass) |
| Final Class | Class cannot be extended | No |
| Final Parameter | Parameter cannot be changed inside method | No |

**Important Notes:**

* Final methods improve performance as they are resolved at compile-time.
* Final variables must be initialized at declaration or in constructor.
* Final classes like java.lang.String are immutable and secure.

**Real-world Example:**

final class BankAccount {

private final String accountNumber;

private double balance;

BankAccount(String accNum, double bal) {

this.accountNumber = accNum;

this.balance = bal;

}

final void deposit(double amount) {

balance += amount;

}

final void showBalance() {

System.out.println("Balance: " + balance);

}

// accountNumber cannot be changed after initialization

String getAccountNumber() {

return accountNumber;

} }

* **Introduction to OOPs Principles**

Object-Oriented Programming (OOP) is a fundamental concept in modern programming that focuses on using objects and classes to structure software. It helps in writing modular, reusable, and maintainable code.

**1. Encapsulation**

* Combines data (variables) and methods (functions) into a single unit called a class.
* Restricts direct access to some of an object’s components, enhancing security and integrity.
* Achieved using access modifiers: private, public, protected.
* Data can only be accessed and modified through getter and setter methods.
* Prevents accidental modification of data (data hiding).
* Improves code modularity and maintainability.

**2. Inheritance**

* Mechanism where one class acquires the properties (fields) and behaviors (methods) of another class.
* Promotes code reuse and reduces redundancy.
* Supports hierarchical classification.
* The class that inherits is called the child/subclass, and the class being inherited from is the parent/superclass.
* Uses the extends keyword in languages like Java.
* Allows method overriding (child class redefines a method from parent class).
* Helps achieve runtime polymorphism.

**3. Polymorphism**

* Allows one interface to be used for a general class of actions.
* Provides the ability to call the same method on different objects and have each respond in its own way.
* Two main types:
  + **Compile-time polymorphism (Method Overloading):** Multiple methods with the same name but different parameters.
  + **Runtime polymorphism (Method Overriding):** Subclass provides specific implementation of a method already defined in parent class.
* Enhances flexibility and maintainability.
* Simplifies code readability and scalability.

**4. Abstraction**

* Hides complex internal logic and shows only the relevant information to the user.
* Achieved using abstract classes and interfaces.
* Encourages focus on essential qualities of an object rather than the background details.
* Promotes cleaner code and reduces implementation complexity.
* Supports implementation of loosely coupled components.

**Additional Advantages of OOP**

* **Modularity:** Code is divided into separate modules (classes) that are easier to manage.
* **Reusability:** Once a behavior is defined, it can be reused in other parts of the application.
* **Scalability:** Object-oriented code is easier to expand and modify.
* **Maintainability:** Bugs can be fixed more easily, and code enhancements are less risky.
* **Security:** Through encapsulation and abstraction, sensitive data is protected.

**Conclusion**  
The four key principles of OOP—Encapsulation, Inheritance, Polymorphism, and Abstraction—serve as the core of object-oriented programming. Mastery of these principles enables developers to build robust, scalable, and maintainable applications with clean and reusable code structures.

1. **Encapsulation in Java**

Encapsulation is one of the core concepts of Object-Oriented Programming (OOP). It is the mechanism of wrapping the data (variables) and the code (methods) that operate on the data into a single unit called a class. It helps in data hiding and controlling access to class members using access modifiers.

**1. Data Hiding**

Encapsulation hides the internal state of an object and requires all interaction to be performed through an object's methods.

**Example:**

class BankAccount {

private double balance = 1000.00; // data is hidden using private

// Getter method

public double getBalance() {

return balance;

}

// Setter method with validation

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

} else {

System.out.println("Invalid deposit amount");

}

}

}

**2. Access Modifiers**

Java provides four types of access modifiers to control access to class members:

1. **Public** – Accessible from anywhere.
2. **Protected** – Accessible within the same package and subclasses.
3. **Default** – Accessible only within the same package.
4. **Private** – Accessible only within the class.

**Example:**

public class Student {

private int rollNumber; // private variable

protected String department; // protected variable

String name; // default access

public int age; // public variable

public void displayInfo() {

System.out.println("Roll Number: " + rollNumber);

System.out.println("Department: " + department);

System.out.println("Name: " + name);

System.out.println("Age: " + age);

}

}

**3. Getter and Setter Methods**

Encapsulation makes use of getter and setter methods to access private fields in a controlled way.

**Example:**

class Employee {

private String employeeId;

private double salary;

public String getEmployeeId() {

return employeeId;

}

public void setEmployeeId(String employeeId) {

this.employeeId = employeeId;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

if (salary >= 0) {

this.salary = salary;

} else {

System.out.println("Salary cannot be negative.");

}

}

}

**4. Maintains Integrity**

Encapsulation ensures that the internal representation of an object is hidden, protecting the object from unintended interference and misuse.

**Example with validation:**

class TemperatureSensor {

private double temperature;

public void setTemperature(double temperature) {

if (temperature >= -50 && temperature <= 100) {

this.temperature = temperature;

} else {

System.out.println("Temperature out of range!");

}

}

public double getTemperature() {

return temperature;

}

}

**5. Enhances Modularity**

Each class is self-contained. Changes to one class don’t affect others as long as the public interface remains consistent.

**Example:**

class Engine {

public void start() {

System.out.println("Engine started");

}

}

class Car {

private Engine engine = new Engine();

public void drive() {

engine.start();

System.out.println("Car is moving");

}

}

**Import & Packages in Java**

**1. Package Definition**

A **package** in Java is a namespace that organizes a set of related classes and interfaces. It helps prevent class name conflicts.

**2. Package Declaration**

Packages are declared at the top of the Java source file:

package com.company.project;

**3. Import Statement**

Used to refer to classes from other packages:

import java.util.Scanner; // Single class

import java.util.\*; // All classes from util

**4. Types of Import**

* **Single-Type Import**: import java.util.List;
* **On-Demand Import**: import java.util.\*;

**5. Avoiding Name Collisions**

Packages help organize classes and avoid name conflicts between classes with the same name in different packages.

**Example:**

package com.myapp.utils;

public class Logger {

public static void log(String message) {

System.out.println("LOG: " + message);

}

}

import com.myapp.utils.Logger;

public class Main {

public static void main(String[] args) {

Logger.log("Application started");

}

}

**Access Modifiers Detailed**

**1. Types of Modifiers:**

* **Public**: Accessible everywhere.
* **Protected**: Accessible within same package and subclasses.
* **Default (No modifier)**: Accessible within same package only.
* **Private**: Accessible only within the class.

**2. Class-Level Access**

* Only public and default can be used.

**3. Member-Level Access**

* Fields, methods, and constructors can use all four modifiers.

**Example:**

public class Vehicle {

private String model;

protected int speed;

String color; // default

public void run() {

System.out.println("Vehicle is running");

}

}

**Getter and Setter**

**1. Getters**

Return the value of a private field.

public String getModel() {

return model;

}

**2. Setters**

Set or update the value of a private field.

public void setModel(String model) {

this.model = model;

}

**3. Control and Validation**

Used for restricting or validating values before setting.

public void setSpeed(int speed) {

if (speed >= 0) {

this.speed = speed;

} else {

System.out.println("Speed cannot be negative.");

}

}

**4. Read-Only/Write-Only**

* **Read-Only Example:**

public int getSpeed() {

return speed;

}

* **Write-Only Example:**

public void setSecretCode(String code) {

this.secretCode = code;

}

**5. Flexibility**

Internal logic can change without affecting the external interface.

public void setDiscount(double discount) {

// Business logic may change later

this.discount = discount > 0 ? discount : 0;

}

Encapsulation is essential for creating clean, maintainable, and secure Java applications. It hides complexity and provides a clear interface for interaction.