**Unit-1 Notes**

**Java Platforms Overview**

Java has evolved into several distinct platforms, each catering to different types of applications and environments. The three main Java platforms are J2SE (Java 2 Standard Edition), J2EE (Java 2 Enterprise Edition), and J2ME (Java 2 Micro Edition). Understanding when and why to use each of these platforms is crucial for selecting the right tools and technologies for your development needs.

**1. J2SE (Java 2 Standard Edition)**

**J2SE**, now referred to as **Java SE (Standard Edition)**, provides the core functionality needed for developing general-purpose applications. It includes the basic Java libraries, APIs, and tools required for building desktop and server-side applications.

**When to Use J2SE:**

* **Desktop Applications**: For creating standalone applications with graphical user interfaces (GUIs) using Swing or JavaFX.
* **Basic Utilities**: For developing utilities, tools, and smaller applications that do not require extensive enterprise features.
* **Server-Side Development**: For building server-side components that are not part of a larger enterprise system.

**Key Components:**

* **Java Virtual Machine (JVM)**: Executes Java bytecode and provides cross-platform capabilities.
* **Core Libraries**: Includes classes for I/O, networking, data structures, and concurrency.
* **Swing/JavaFX**: Libraries for building graphical user interfaces (GUIs).

**Development Examples:**

* **Desktop GUI Applications**: E.g., a personal finance manager or a text editor.
* **Utilities**: E.g., file managers or data processing tools.
* **Server Applications**: E.g., basic server-side applications for handling HTTP requests.

**2. J2EE (Java 2 Enterprise Edition)**

**J2EE**, now known as **Java EE (Enterprise Edition)**, is designed for developing large-scale, distributed, and enterprise-level applications. It extends J2SE by providing additional libraries, APIs, and tools for building enterprise-grade applications.

**When to Use J2EE:**

* **Enterprise Applications**: For building large-scale applications with complex business logic and multiple layers, such as ERP systems or CRM systems.
* **Web Applications**: For developing web-based applications using Servlets, JSP, and JavaServer Faces (JSF).
* **Distributed Systems**: For creating applications that need to be distributed across multiple servers or locations using technologies like Enterprise JavaBeans (EJB) and Java Message Service (JMS).

**Key Components:**

* **Servlets and JSP**: For creating dynamic web content and handling requests.
* **EJB**: For business logic and transaction management.
* **JMS**: For messaging between distributed components.
* **JPA**: For object-relational mapping and database interaction.

**Development Examples:**

* **Enterprise Resource Planning (ERP) Systems**: E.g., managing business operations and resources.
* **Customer Relationship Management (CRM) Systems**: E.g., handling customer interactions and data.
* **E-Commerce Platforms**: E.g., online shopping systems.

**3. J2ME (Java 2 Micro Edition)**

**J2ME**, now referred to as **Java ME (Micro Edition)**, is designed for mobile and embedded devices with limited resources. It provides a subset of Java SE's libraries and APIs tailored for small devices such as mobile phones, PDAs, and IoT devices.

**When to Use J2ME:**

* **Mobile Applications**: For developing applications that run on mobile phones or PDAs.
* **Embedded Systems**: For building applications for devices with limited processing power and memory.
* **IoT Devices**: For creating applications for Internet of Things (IoT) devices that have restricted resources.

**Key Components:**

* **MIDP (Mobile Information Device Profile)**: For building mobile applications.
* **CLDC (Connected Limited Device Configuration)**: Provides the minimal runtime environment for constrained devices.
* **APIs**: Specialized APIs for mobile and embedded devices.

**Development Examples:**

* **Mobile Games**: E.g., simple games for feature phones.
* **Utility Applications**: E.g., weather apps or calculators for mobile devices.
* **Embedded Systems**: E.g., applications for smart sensors or small appliances.

**Summary**

* **J2SE (Java SE)**: Use for general-purpose programming, including desktop and basic server applications. Provides core Java libraries and tools.
* **J2EE (Java EE)**: Use for enterprise-level applications, web applications, and distributed systems. Extends J2SE with additional enterprise-focused libraries and APIs.
* **J2ME (Java ME)**: Use for mobile and embedded systems with limited resources. Provides a subset of Java SE tailored for small devices.

### Q. What is compiler and Interprator?

### Compiler

A **compiler** is a program that translates the entire source code of a high-level programming language into machine code or an intermediate form in one go. This process is called compilation.

#### How Compilers Work:

1. **Source Code**: The programmer writes code in a high-level language (e.g., Java, C++), which is then fed into the compiler.
2. **Compilation**: The compiler analyzes and translates the entire source code into machine code or bytecode (an intermediate representation) in a single operation.
3. **Output**: The result is typically an executable file or bytecode file that can be run directly by the operating system or virtual machine.

#### Advantages:

* **Performance**: Programs compiled into machine code usually execute faster because the translation is done beforehand.
* **Optimization**: Compilers can optimize the code to improve performance and efficiency.

#### Disadvantages:

* **Compilation Time**: The entire code needs to be compiled before execution, which can be time-consuming.
* **Debugging**: Errors must be fixed in the source code and then recompiled, which can be less interactive than interpreted languages.

#### Examples:

* **C/C++**: These languages are typically compiled into machine code.
* **Java**: Java code is compiled into bytecode, which is then executed by the Java Virtual Machine (JVM).

### Interpreter

An **interpreter** is a program that translates and executes source code line-by-line or statement-by-statement. It reads the source code and performs the instructions directly.

#### How Interpreters Work:

1. **Source Code**: The programmer writes code in a high-level language.
2. **Interpretation**: The interpreter reads the source code line-by-line or statement-by-statement and executes it directly.
3. **Execution**: The result is immediate execution of the code, but without producing a separate executable file.

#### Advantages:

* **Ease of Debugging**: Errors can be identified and corrected in real-time, as the code is executed line-by-line.
* **No Compilation Needed**: The source code can be executed directly, which can simplify development and testing.

#### Disadvantages:

* **Performance**: Interpreted code generally runs slower than compiled code because the translation happens at runtime.
* **Runtime Overhead**: The interpreter must be available during execution, which can add overhead.

#### Examples:

* **Python**: Python is typically interpreted, meaning the Python interpreter executes the code line-by-line.
* **JavaScript**: JavaScript code is interpreted by web browsers to execute within the context of a webpage.

### Key Differences

| **Aspect** | **Compiler** | **Interpreter** |
| --- | --- | --- |
| **Translation** | Translates entire code at once | Translates code line-by-line or statement-by-statement |
| **Execution** | Produces an executable file or bytecode | Executes code directly without producing an intermediate file |
| **Performance** | Generally faster execution | Generally slower execution due to real-time translation |
| **Debugging** | Requires recompilation for errors | Easier debugging with real-time error reporting |
| **Development Cycle** | Longer initial setup but faster execution | Immediate execution, quicker to test small code changes |

### Conclusion

* **Compilers** are suited for scenarios where performance is critical and you want to optimize the final executable.
* **Interpreters** are useful for development and debugging, where ease of modification and immediate feedback are more important than execution speed.

### Q. Standalone vs Distributed application?

### Standalone Application

**Standalone applications** are self-contained software programs that run independently on a single machine or device. They do not require a network connection to operate, and all necessary resources are bundled within the application itself or are locally available.

#### Characteristics:

* **Single-User**: Typically designed for use by a single user at a time.
* **Local Resources**: All resources (data, files, etc.) are stored and accessed locally on the machine where the application is installed.
* **No Network Dependency**: Does not require network connectivity to function.
* **Simpler Architecture**: Generally simpler in terms of design and development compared to distributed applications.

#### Advantages:

* **Offline Capability**: Can be used without an internet connection.
* **Performance**: Often faster in terms of response time since it does not rely on network communication.
* **Simplicity**: Easier to develop and deploy because it does not involve complex network interactions or server management.

#### Disadvantages:

* **Limited Scalability**: Not designed for handling multiple users or large-scale operations.
* **Data Synchronization**: Data stored locally is not automatically synchronized with other systems or users.
* **Update Management**: Updates must be installed on each individual machine.

#### Examples:

* **Desktop Applications**: Microsoft Word, Adobe Photoshop.
* **Utilities**: System utilities, local database management tools.

### Distributed Application

**Distributed applications** are software systems that run on multiple interconnected machines or nodes, often across a network. They are designed to work together to provide a cohesive service or functionality.

#### Characteristics:

* **Multi-User**: Supports multiple users or clients simultaneously.
* **Network-Based**: Requires network connectivity to communicate between different components or nodes.
* **Scalable Architecture**: Can be scaled horizontally by adding more nodes or servers to handle increased load.
* **Complex Interactions**: Involves coordination and communication between different parts of the system.

#### Advantages:

* **Scalability**: Can be scaled to handle more users or larger data volumes by adding more servers or nodes.
* **Reliability**: Can be designed to be fault-tolerant, with redundancy and failover mechanisms.
* **Centralized Management**: Allows for centralized data storage and management, which can be accessed from multiple locations.

#### Disadvantages:

* **Complexity**: More complex to design, develop, and deploy due to the need for network communication and coordination.
* **Latency**: Network communication can introduce latency and affect performance.
* **Security**: Requires robust security measures to protect data and communication over the network.

#### Examples:

* **Web Applications**: Online banking systems, social media platforms.
* **Enterprise Systems**: Customer Relationship Management (CRM) systems, Enterprise Resource Planning (ERP) systems.

### Summary

* **Standalone Applications**: Self-contained, run on a single machine, do not require network connectivity. Examples include desktop applications and utilities.
* **Distributed Applications**: Run across multiple machines, require network connectivity, designed for scalability and multi-user support. Examples include web applications and enterprise systems.

### Q. Monolithic vs Microservice Application?

### Monolithic Architecture

**Monolithic Architecture** is a traditional model where an application is developed as a single, unified unit. All components of the application are interwoven into a single codebase and deployed together.

#### Characteristics:

* **Single Codebase**: All functionalities are contained within a single codebase.
* **Tightly Coupled**: Components are closely interconnected, making changes in one area potentially affect others.
* **Unified Deployment**: The entire application is packaged and deployed as a single unit.

#### Advantages:

1. **Simplicity**: Easier to develop, test, and deploy since all components are contained within a single codebase.
2. **Performance**: Can be more performant for small to medium-sized applications as there is no inter-service communication overhead.
3. **Easier to Debug**: Debugging is simpler as there is a single application and a single codebase to analyze.

#### Disadvantages:

1. **Scalability**: Difficult to scale specific components individually. Scaling the application requires scaling the entire monolith.
2. **Complexity Over Time**: As the application grows, it becomes more complex and harder to maintain, leading to a "big ball of mud" scenario.
3. **Deployment Challenges**: Any change or update requires redeploying the entire application, which can be risky and time-consuming.

#### Example:

Consider an e-commerce application where the user interface, product catalog, and order processing are all part of a single, large application.

### Microservices Architecture

**Microservices Architecture** is a design approach where an application is broken down into a set of loosely coupled, independently deployable services. Each microservice focuses on a specific business capability and communicates with others via well-defined APIs.

#### Characteristics:

* **Decomposed Services**: The application is divided into multiple microservices, each responsible for a distinct functionality.
* **Loose Coupling**: Services are loosely coupled, meaning changes in one service have minimal impact on others.
* **Independent Deployment**: Each microservice can be developed, deployed, and scaled independently.

#### Advantages:

1. **Scalability**: Individual services can be scaled independently based on demand, improving resource utilization.
2. **Flexibility**: Different technologies, frameworks, and programming languages can be used for different services.
3. **Resilience**: Failures in one service do not necessarily bring down the entire application. This can lead to improved system reliability.
4. **Team Autonomy**: Development teams can work on different services independently, improving development speed and efficiency.

#### Disadvantages:

1. **Complexity**: Increased complexity in managing multiple services, including communication, data consistency, and deployment.
2. **Inter-Service Communication**: Communication between services can introduce latency and requires handling various issues like data serialization, network failures, etc.
3. **Data Management**: Data consistency and transactions across services can be challenging to manage.

#### Example:

In an e-commerce application using microservices, you might have separate services for user management, product catalog, order processing, and payment. Each service runs independently and interacts with others through APIs.

### Comparison

| **Aspect** | **Monolithic Architecture** | **Microservices Architecture** |
| --- | --- | --- |
| **Structure** | Single unified codebase | Multiple loosely coupled services |
| **Deployment** | Deploy as a single unit | Deploy each service independently |
| **Scalability** | Scale the entire application | Scale individual services |
| **Complexity** | Simpler for small applications | More complex due to service orchestration |
| **Flexibility** | Less flexibility in technology choices | Greater flexibility in technology choices |
| **Resilience** | Failure affects the entire application | Failure in one service does not bring down others |
| **Development** | All features developed together | Features can be developed by separate teams |

### Choosing Between Monolithic and Microservices

* **Monolithic Architecture** is often suitable for small to medium-sized applications with a single team where simplicity and ease of deployment are prioritized.
* **Microservices Architecture** is ideal for large-scale applications requiring scalability, flexibility, and independent development teams.

### Q. Conditional Statements in Java

Conditional statements in Java are used to execute specific blocks of code based on the evaluation of conditions. They are fundamental in controlling the flow of execution in a program. Here’s a comprehensive overview:

#### 1. **if Statement**

**Definition:** The if statement executes a block of code if a specified condition is true.

**Syntax:**

if (condition) {

// Code to be executed if condition is true

}

**Example:**

int age = 20;

if (age >= 18) {

System.out.println("You are an adult.");

}

**When to Use:** Use the if statement when you need to execute a block of code based on a single condition.

**Real-Time Use Case:** Checking if a user’s age qualifies them for adult content access or verifying if an input value meets certain criteria.

#### 2. **if-else Statement**

**Definition:** The if-else statement provides an alternative block of code to be executed if the condition is false.

**Syntax:**

if (condition) {

// Code to be executed if condition is true

} else {

// Code to be executed if condition is false

}

**Example:**

int age = 16;

if (age >= 18) {

System.out.println("You are an adult.");

} else {

System.out.println("You are not an adult.");

}

**When to Use:** Use the if-else statement when you need to handle two mutually exclusive cases.

**Real-Time Use Case:** Deciding whether to grant access based on user authentication or handling valid vs. invalid input.

#### 3. **if-else-if Ladder**

**Definition:** The if-else-if ladder allows for multiple conditions to be checked sequentially. The first true condition’s block is executed, and the rest are ignored.

**Syntax:**

if (condition1) {

// Code to be executed if condition1 is true

} else if (condition2) {

// Code to be executed if condition2 is true

} else if (condition3) {

// Code to be executed if condition3 is true

} else {

// Code to be executed if none of the above conditions are true

}

**Example:**

int score = 85;

if (score >= 90) {

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

} else if (score >= 80) {

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

} else if (score >= 70) {

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

} else {

System.out.println("Grade: D");

}

**When to Use:** Use the if-else-if ladder when you need to evaluate multiple conditions and execute different blocks of code based on which condition is true.

**Real-Time Use Case:** Assigning grades based on score ranges or categorizing items based on their attributes.

### 4*. Nested if Statements in Java*

**Definition:** Nested if statements are if statements placed inside other if statements, allowing for more complex decision-making.

**Syntax:**

if (condition1) {

if (condition2) {

// Code executed if both conditions are true

}

}

**Example:**

int age = 20;

boolean hasTicket = true;

if (age >= 18) { // Check if age is 18 or older

if (hasTicket) { // Check if ticket is available

System.out.println("You can enter the movie."); // Both conditions true

} else {

System.out.println("You need a ticket to enter."); // Age is 18+, but no ticket

}

} else {

System.out.println("You are not old enough to enter."); // Age less than 18

}

**Working:**

1. **Outer if**: Checks if age >= 18.
2. **Inner if**: If the outer condition is true, it checks if hasTicket is true.
3. **Output**:
   * Both conditions true: "You can enter the movie."
   * Outer true, inner false: "You need a ticket to enter."
   * Outer false: "You are not old enough to enter."

**When to Use:**

* For evaluating multiple, dependent conditions.
* When one condition must be true for the next condition to be checked.

**Real-Time Use Case:**

* Determining eligibility based on multiple criteria, like age and ticket possession for event entry.

#### 5. **Ternary Operator**

**Definition:** The ternary operator is a shorthand for the if-else statement. It is used to evaluate a condition and choose between two values based on that condition.

**Syntax:**

condition ? value\_if\_true : value\_if\_false;

**Example:**

int age = 20;

String result = (age >= 18) ? "Adult" : "Not an adult";

System.out.println(result);

**When to Use:** Use the ternary operator for simple conditional assignments. It makes the code concise and easier to read for straightforward cases.

**Real-Time Use Case:** Assigning values based on a condition in a concise manner, such as setting user roles or status messages.

### Summary

* **if Statement:** Use when you need to execute code based on a single condition.
* **if-else Statement:** Use when you need to handle two mutually exclusive cases.
* **if-else-if Ladder:** Use when you have multiple conditions to evaluate sequentially.
* **Nested if Statement:** Use for complex decision-making involving multiple layers of conditions.
* **Ternary Operator:** Use for concise conditional assignments.

**Q. Difference between Switch vs for, while and do-while loop?**

The **main difference** between a switch statement and a loop (for, while, do-while) in Java lies in their **purpose and behavior**:

**1. Purpose:**

* **Switch Case:**
  + A switch statement is used for **decision-making** where you want to execute different blocks of code based on the value of a single expression or variable.
  + It **selects** one of many possible **code paths** to execute, depending on the value of the expression.
  + **No iteration** or repetition happens here.
* **Loop:**
  + A loop (for, while, do-while) is used for **repeating** a block of code **multiple times** until a specified condition is met.
  + Loops are used when you need to **execute the same block of code multiple times**, either a fixed number of times or until a condition becomes false.

**2. Execution Flow:**

* **Switch Case:**
  + It **evaluates a single expression** (like a variable or result of an operation) and matches it against different cases. The code in the matching case is executed.
  + It is a **one-time selection** and doesn't repeat unless explicitly called again.
* **Loop:**
  + The loop **repeatedly executes** the same block of code as long as a specified condition is true.
  + It **keeps executing** until the condition is false or a break is encountered.

**3. Syntax Differences:**

* **Switch Case Example:**

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");

}

**Explanation:**

* + Here, based on the value of day, the corresponding case block executes. It’s a **single choice** operation based on the value of day.
* **Loop Example:**

for (int i = 1; i <= 5; i++) {

System.out.println("Iteration: " + i);

}

**Explanation:**

* + This loop will execute 5 times, incrementing the value of i each time. It **repeats** the same block of code until the condition i <= 5 is no longer true.

**4. Use Cases:**

* **Switch Case:**
  + **Used when**: You have a variable with multiple possible values, and you want to execute different code based on each value.
  + **Example**: Menu selection, handling multiple states (e.g., days of the week, grades).
* **Loop:**
  + **Used when**: You need to perform **repetitive tasks** like iterating over arrays, processing input multiple times, or running code until a condition is met.
  + **Example**: Iterating over a list of items, performing a task repeatedly, counting occurrences.

**5. Repetition and Iteration:**

* **Switch Case:**
  + **No repetition**. Once a case is selected, only that block of code runs. You **cannot iterate** through cases.
* **Loop:**
  + **Repetition is inherent**. The same block of code runs multiple times as long as the condition is satisfied.

**Summary:**

| **Feature** | **Switch Case** | **Loop** |
| --- | --- | --- |
| **Purpose** | Decision making based on a single value | Repeating a block of code multiple times |
| **Condition** | Single value evaluation | Condition checked before or after each iteration |
| **Repetition** | No repetition, executes once per case | Repeats as long as the condition is true |
| **Use Case** | Menu selections, handling multiple cases | Iterating over arrays, processing data repeatedly |
| **Example** | Handling different days of the week | Counting numbers, looping through a list |

### 1. ****For Loop****

#### When to use:

* When you know in advance **how many times** the loop needs to run (i.e., a fixed number of iterations).
* Ideal for iterating through arrays, lists, or collections where the number of elements is known.

#### Syntax:

for (initialization; condition; increment/decrement) {

// Code to be executed

}

#### Real-Time Use Case:

**Scenario**: Printing a list of student names.

public class ForLoopExample {

public static void main(String[] args) {

String[] students = {"Alice", "Bob", "Charlie", "David"};

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

System.out.println("Student: " + students[i]);

}

}

}

Here, you know the number of students in advance, so the for loop is a perfect fit.

### 2. ****While Loop****

#### When to use:

* When you **do not know** how many iterations are required but you have a **condition** that needs to be checked before entering the loop.
* Ideal for scenarios where you need to keep repeating an action until a certain condition is met.

#### Syntax:

while (condition) {

// Code to be executed

}

#### Real-Time Use Case:

**Scenario**: Taking user input until they enter the correct password.

import java.util.Scanner;

public class WhileLoopExample {

public static void main(String[] args) {

String correctPassword = "java123";

Scanner scanner = new Scanner(System.in);

String input;

System.out.print("Enter password: ");

input = scanner.nextLine();

while (!input.equals(correctPassword)) {

System.out.print("Incorrect password. Try again: ");

input = scanner.nextLine();

}

System.out.println("Access granted!");

}

}

Here, you don’t know how many attempts the user will take to enter the correct password, so a while loop is ideal.

### 3. ****Do-While Loop****

#### When to use:

* When you need the loop to run **at least once**, regardless of the condition, and then repeat as long as the condition is true.
* Ideal for user interaction scenarios where you want to ensure an action is performed at least once.

#### Syntax:

do {

// Code to be executed

} while (condition);

#### Real-Time Use Case:

**Scenario**: Displaying a menu until the user selects the "Exit" option.

import java.util.Scanner;

public class DoWhileLoopExample {

public static void main(String[] args) {

int choice;

Scanner scanner = new Scanner(System.in);

do {

System.out.println("Menu:");

System.out.println("1. Start Game");

System.out.println("2. Load Game");

System.out.println("3. Exit");

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

choice = scanner.nextInt();

} while (choice != 3);

System.out.println("Exiting the game...");

}

}

Here, the menu is displayed **at least once**, and the loop continues until the user selects "Exit." This makes do-while suitable since you want the menu to show at least once.

### Summary:

* **For Loop**: Use when the number of iterations is known beforehand (e.g., iterating over arrays, collections).
* **While Loop**: Use when you do not know how many iterations are required but need to check the condition **before** entering the loop (e.g., user validation, fetching data).
* **Do-While Loop**: Use when you want to ensure the loop runs **at least once** regardless of the condition, and then repeat based on the condition (e.g., displaying menus, user prompts).

Each type of loop fits specific real-world programming scenarios depending on whether you know the number of iterations and when the condition should be checked.

### Q. Java Operators

In Java, operators are symbols that perform operations on variables and values. They can be categorized into several types:

#### 1. Unary Operator

**Definition:** Unary operators operate on a single operand to produce a new value.

**Types and Syntax:**

* **Unary Plus (+)**: Indicates a positive value.

int a = +5; // a is 5

* **Unary Minus (-)**: Negates the value.

int b = -5; // b is -5

* **Increment (++)**: Increases the value by 1.

int c = 10;

c++; // c is now 11

* **Decrement (--)**: Decreases the value by 1.

int d = 10;

d--; // d is now 9

* **Logical NOT (!)**: Reverses the logical state.

boolean e = true;

boolean f = !e; // f is false

**Practical Example:**

int x = 5;

x++; // x is 6

System.out.println("Incremented value: " + x);

boolean flag = true;

System.out.println("Negated value: " + !flag); // false

#### 2. Arithmetic Operator

**Definition:** Arithmetic operators perform mathematical operations.

**Types and Syntax:**

* **Addition (+)**: Adds two values.

int sum = 5 + 3; // sum is 8

* **Subtraction (-)**: Subtracts the second value from the first.

int difference = 5 - 3; // difference is 2

* **Multiplication (\*)**: Multiplies two values.

int product = 5 \* 3; // product is 15

* **Division (/)**: Divides the first value by the second.

int quotient = 5 / 2; // quotient is 2

* **Modulus (%)**: Finds the remainder of division.

int remainder = 5 % 2; // remainder is 1

**Practical Example:**

int a = 10;

int b = 3;

System.out.println("Sum: " + (a + b)); // 13

System.out.println("Product: " + (a \* b)); // 30

System.out.println("Quotient: " + (a / b)); // 3

System.out.println("Remainder: " + (a % b)); // 1

#### 3. Shift Operator

**Definition:** Shift operators shift the bits of a value left or right.

**Types and Syntax:**

* **Left Shift (<<)**: Shifts bits to the left, filling with zero.

int a = 5 << 1; // a is 10 (binary 0101 shifted to 1010)

* **Right Shift (>>)**: Shifts bits to the right, preserving the sign bit.

int b = 5 >> 1; // b is 2 (binary 0101 shifted to 0010)

* **Unsigned Right Shift (>>>)**: Shifts bits to the right, filling with zero regardless of sign.

int c = -5 >>> 1; // c is a large positive value

**Practical Example:**

int number = 16;

System.out.println("Left Shift: " + (number << 2)); // 64

System.out.println("Right Shift: " + (number >> 2)); // 4

#### 4. Relational Operator

**Definition:** Relational operators compare two values and return a boolean result.

**Types and Syntax:**

* **Equal to (==)**: Checks if two values are equal.

boolean result = (5 == 5); // true

* **Not Equal to (!=)**: Checks if two values are not equal.

boolean result = (5 != 3); // true

* **Greater Than (>)**: Checks if the first value is greater than the second.

boolean result = (5 > 3); // true

* **Less Than (<)**: Checks if the first value is less than the second.

boolean result = (5 < 8); // true

* **Greater Than or Equal to (>=)**: Checks if the first value is greater than or equal to the second.

boolean result = (5 >= 5); // true

* **Less Than or Equal to (<=)**: Checks if the first value is less than or equal to the second.

boolean result = (5 <= 8); // true

**Practical Example:**

int a = 10;

int b = 20;

System.out.println("a == b: " + (a == b)); // false

System.out.println("a < b: " + (a < b)); // true

#### 5. Bitwise Operator

### 1. ****Bitwise AND (****&****)****

* The **bitwise AND** operator compares each corresponding bit of two numbers and returns 1 if **both** bits are 1, otherwise it returns 0.

#### Example:

int a = 5 & 3;

System.out.println(a); // Output: 1

**Explanation**:

1. Convert the numbers to binary:
   * 5 in binary is 0101.
   * 3 in binary is 0011.
2. Perform the AND operation on each bit:

0101 (binary for 5)

& 0011 (binary for 3)

------

0001 (result of AND)

1. The result is 0001 in binary, which equals 1 in decimal.

### 2. ****Bitwise OR (****|****)****

* The **bitwise OR** operator compares each corresponding bit of two numbers and returns 1 if **either** bit is 1, otherwise it returns 0.

#### Example:

int b = 5 | 3;

System.out.println(b); // Output: 7

**Explanation**:

1. Convert the numbers to binary:
   * 5 in binary is 0101.
   * 3 in binary is 0011.
2. Perform the OR operation on each bit:

0101 (binary for 5)

| 0011 (binary for 3)

------

0111 (result of OR)

1. The result is 0111 in binary, which equals 7 in decimal.

### 3. ****Bitwise XOR (****^****)****

* The **bitwise XOR** (exclusive OR) operator compares each corresponding bit of two numbers and returns 1 if the bits are **different**, otherwise it returns 0.

#### Example:

int c = 5 ^ 3;

System.out.println(c); // Output: 6

**Explanation**:

1. Convert the numbers to binary:
   * 5 in binary is 0101.
   * 3 in binary is 0011.
2. Perform the XOR operation on each bit:

0101 (binary for 5)

^ 0011 (binary for 3)

------

0110 (result of XOR)

1. The result is 0110 in binary, which equals 6 in decimal.

### 4. ****Bitwise Complement (****~****)****

* The **bitwise complement** operator (~) inverts all the bits of the number. In Java, numbers are stored using **two's complement** for negative numbers.

#### Example:

int d = ~5;

System.out.println(d); // Output: -6

**Explanation**:

1. Convert 5 to binary (32 bits):
   * 5 in binary (32-bit) is 00000000 00000000 00000000 00000101.
2. Perform the bitwise NOT (invert all the bits):

~00000000 00000000 00000000 00000101

= 11111111 11111111 11111111 11111010

1. This result represents -6 in two's complement:
   * Inverting all bits gives us 00000000 00000000 00000000 00000110 (which is 6), but in two's complement, this becomes -6.

### ****Summary of Operators****:

1. **AND (&)**: Both bits must be 1 to get 1.
   * Example: 5 & 3 → 0101 & 0011 = 0001 → Result: 1.
2. **OR (|)**: Either bit must be 1 to get 1.
   * Example: 5 | 3 → 0101 | 0011 = 0111 → Result: 7.
3. **XOR (^)**: If the bits are different, the result is 1.
   * Example: 5 ^ 3 → 0101 ^ 0011 = 0110 → Result: 6.
4. **Complement (~)**: Inverts all bits (using two's complement for negative numbers).
   * Example: ~5 → Inverts 00000000 00000000 00000000 00000101 to 11111111 11111111 11111111 11111010, which is -6.

#### 6. Logical Operator

**Definition:** Logical operators perform logical operations on boolean values.

**Types and Syntax:**

* **Logical AND (&&)**: Returns true if both conditions are true.

boolean result = (true && false); // false

* **Logical OR (||)**: Returns true if at least one condition is true.

boolean result = (true || false); // true

* **Logical NOT (!)**: Reverses the boolean value.

boolean result = !true; // false

**Practical Example:**

boolean a = true;

boolean b = false;

System.out.println("AND: " + (a && b)); // false

System.out.println("OR: " + (a || b)); // true

System.out.println("NOT: " + !a); // false

#### 7. Ternary Operator

**Definition:** The ternary operator is a shorthand for if-else statements, returning one of two values based on a condition.

**Syntax:**

condition ? valueIfTrue : valueIfFalse;

**Example:**

int a = 10;

int b = 20;

int max = (a > b) ? a : b; // max is 20

**Practical Example:**

int age = 18;

String status = (age >= 18) ? "Adult" : "Minor";

System.out.println("Status: " + status); // Adult

#### 8. Assignment Operator

**Definition:** Assignment operators assign values to variables.

**Types and Syntax:**

* **Simple Assignment (=)**: Assigns a value to a variable.

int a = 5;

* **Add and Assign (+=)**: Adds and assigns a value.

a += 3; // a is 8

* **Subtract and Assign (-=)**: Subtracts and assigns a value.

a -= 2; // a is 6

* **Multiply and Assign (\*=)**: Multiplies and assigns a value.

a \*= 4; // a is 24

* **Divide and Assign (/=)**: Divides and assigns a value.

a /= 3; // a is 8

* **Modulus and Assign (%=)**: Computes modulus and assigns a value.

a %= 5; // a is 3

**Practical Example:**

int x = 10;

x += 5; // x is 15

x \*= 2; // x is 30

x /= 3; // x is 10

System.out.println("Final value: " + x); // 10

**Q. Java Keywords Breakdown:**

1. **Data Type-Related Keywords (8)**:
   * byte, short, int, long, float, double, char, boolean
2. **Control Flow-Related Keywords (11)**:
   * if, else, switch, case, default, while, do, for, break, continue, return
3. **Object-Related Keywords (5)**:
   * new, instanceof, super, this, null
4. **Return Type Keyword (1)**:
   * void
5. **Exception-Related Keywords (7)**:
   * try, catch, finally, throw, throws, assert, enum (Note: enum also serves as a type)
6. **Class-Related Keywords (7)**:
   * class, interface, extends, implements, package, import, enum
7. **Access Modifier Keywords (3)**:
   * public, private, protected
8. **Non-Access Modifier-Related Keywords (8)**:
   * static, final, abstract, synchronized, native, strictfp, transient, volatile
9. **Other Important Keywords (4)**:
   * true, false, null, const (reserved but not used)
10. **Unused or Reserved Keywords (2)**:

* const, goto (These are reserved for potential future use but are not currently used in Java.)

**Complete Breakdown:**

* **Total keywords**: 53 (51 are in active use; const and goto are reserved and not used, and true, false, and null are considered literals).

Thus, the total number of keywords in active use, excluding the reserved ones (const and goto), is 51-3(literals)=48

### Q. ****Order of Execution of Java Program:****

1. **Static Variables Initialization:**
   * **When:** When the class is first loaded by the JVM.
   * **Details:** Static variables are initialized to their default values (e.g., 0 for int, null for objects) and then assigned their explicit values if provided. This happens before any static blocks are executed.
2. **Static Blocks Execution:**
   * **When:** After static variables are initialized and before any instance of the class is created.
   * **Details:** Static blocks are executed in the order they appear in the class. They are used for initializing static variables or performing setup tasks needed when the class is loaded.
3. **Instance Variables Initialization:**
   * **When:** When an instance of the class is created.
   * **Details:** Instance variables are initialized to their default values and then assigned explicit values defined in their declaration or constructor.
4. **Instance Blocks Execution:**
   * **When:** After instance variables are initialized and before the constructor is executed.
   * **Details:** Instance initialization blocks are executed in the order they appear in the class. They are used for initializing instance variables or performing setup tasks needed for every instance.
5. **Constructor Execution:**
   * **When:** After instance initialization blocks have been executed.
   * **Details:** The constructor initializes new objects. It is called when an instance of the class is created, and it sets up the instance's initial state.
6. **Static Methods:**
   * **When:** Static methods can be called at any time after the class is loaded.
   * **Details:** Static methods can be called without creating an instance of the class. They can access static variables but not instance variables directly.
7. **Instance Methods:**
   * **When:** After an object of the class has been created and initialized.
   * **Details:** Instance methods operate on an instance of the class. They can access both instance variables and static variables.

### ****Execution Flow Example:****

Here is an example to illustrate the order of execution:

public class Example {

// Static variable

static int staticVar = 10;

// Static block

static {

System.out.println("Static block 1 executed");

staticVar = 20;

}

// Another static block

static {

System.out.println("Static block 2 executed");

}

// Instance variable

int instanceVar = 30;

// Instance block

{

System.out.println("Instance block executed");

instanceVar = 40;

}

// Constructor

Example() {

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

instanceVar = 50;

}

// Static method

static void staticMethod() {

System.out.println("Static method executed");

System.out.println("Static variable: " + staticVar);

}

// Instance method

void instanceMethod() {

System.out.println("Instance method executed");

System.out.println("Instance variable: " + instanceVar);

}

public static void main(String[] args) {

// Calling static method without creating an object

Example.staticMethod();

// Creating an object

Example obj = new Example();

obj.instanceMethod();

}

}

### ****Summary:****

* **Static variables** are initialized when the class is loaded.
* **Static blocks** execute after static variables are initialized and before any objects are created.
* **Instance variables** are initialized when an object is created.
* **Instance blocks** execute after instance variables are initialized and before the constructor.
* **Constructors** execute after instance blocks and are used to set up new objects.
* **Static methods** can be called without creating an instance of the class.
* **Instance methods** can be called on an object of the class.

**Another program**

class Example {

// Static variable

static int staticVar = 10;

// Static block

static {

System.out.println("Static block executed");

staticVar = 20;

}

// Instance variable

int instanceVar = 30;

// Instance block

{

System.out.println("Instance block executed");

instanceVar = 40;

}

// Constructor

Example() {

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

instanceVar = 50;

}

// Static method

static void staticMethod() {

System.out.println("Static method executed");

System.out.println("Static variable: " + staticVar);

}

// Instance method

void instanceMethod() {

System.out.println("Instance method executed");

System.out.println("Instance variable: " + instanceVar);

}

public static void main(String[] args) {

// Calling static method without creating an object

Example.staticMethod();

// Creating an object

Example obj = new Example();

obj.instanceMethod();

}

}

**Q. Difference between JDK, JRE and JVM?**

### JDK (Java Development Kit)

* **Definition**: JDK stands for Java Development Kit. It is a software development environment used to develop Java applications. The JDK includes a variety of tools and components essential for Java development.
* **Components**:
  + **Compiler**: Used to compile source code (.java files) into bytecode (.class files).
  + **JRE (Java Runtime Environment)**: Provides libraries and other resources needed to execute Java programs.
  + **Development Tools**: Includes tools such as a debugger, build tools, and documentation generators(docs).

### JRE (Java Runtime Environment)

* **Definition**: JRE stands for Java Runtime Environment. It is the implementation of the JVM (Java Virtual Machine) and provides the necessary environment to run Java applications.
* **Components**:
  + **Java Class Library**: Contains core API classes and packages required for running Java applications.
  + **JVM**: Converts bytecode into machine-specific code and executes it.

### JVM (Java Virtual Machine)

* **Definition**: JVM stands for Java Virtual Machine. It is an abstract machine that provides the runtime environment in which Java bytecode can be executed. The JVM is a specification that defines the working of the Java Virtual Machine.
* **Components**:

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

**Role:** The Class Loader is responsible for loading Java classes into the JVM at runtime. It ensures that the classes needed for execution are loaded from the file system or network into memory.

**Types of Class Loaders:**

1. **Bootstrap Class Loader:**
   * **Role:** The first class loader in the hierarchy. It is part of the core JVM and loads essential classes from the rt.jar file (runtime classes like java.lang.\*, java.util.\*).
   * **Scope:** Loads core Java classes and is implemented in native code.
2. **Extension Class Loader:**
   * **Role:** Loads classes from the jre/lib/ext directory or any directory specified by the java.ext.dirs system property.
   * **Scope:** Handles classes that are part of standard extensions.
3. **Application Class Loader:**
   * **Role:** Also known as the System Class Loader, it loads classes from the classpath specified by the CLASSPATH environment variable or the -cp option.
   * **Scope:** Loads user-defined classes and application-specific libraries.

**Class Loading Process:**

1. **Loading:** The class loader reads the class file from the file system or network and loads it into memory.
2. **Linking:** Consists of:
   * **Verification:** Checks the bytecode for security and correctness to prevent illegal access or modifications.
   * **Preparation:** Allocates memory for class variables and initializes them with default values.
   * **Resolution:** Replaces symbolic references in the bytecode with direct references.
3. **Initialization:** Initializes class variables with their specified values and executes static initializers.

### ****2. Memory Areas:****

**a. Method Area (Class Area):**

* **Role:** Stores class-level data such as class structures (metadata), method data, and static variables.
* **Scope:** Shared among all threads.
* **Details:** Includes the runtime constant pool, field and method data, and method and constructor code.

**b. Heap Area:**

* **Role:** Stores all the objects and their instance variables. This area is where dynamic memory allocation occurs.
* **Scope:** Shared among all threads.
* **Details:** The garbage collector manages this area, reclaiming memory from objects that are no longer reachable.

**c. Stack Area:**

* **Role:** Each thread has its own stack. It stores local variables, method call information, and partial results.
* **Scope:** Thread-specific.
* **Details:** Each method call creates a new stack frame that holds the local variables and method parameters. It also maintains the method's execution context.

**d. PC (Program Counter) Register:**

* **Role:** Contains the address of the currently executing JVM instruction.
* **Scope:** Thread-specific.
* **Details:** Helps track the execution flow of the current thread. Each thread has its own PC register.

**e. Native Method Stack:**

* **Role:** Used for executing native methods (methods written in languages other than Java, such as C or C++).
* **Scope:** Thread-specific.
* **Details:** Handles calls to native libraries and manages their execution.

### ****3. Execution Engine:****

**a. Interpreter:**

* **Role:** Reads and executes bytecode instructions one by one.
* **Scope:** Part of the JVM that deals with bytecode execution.
* **Details:** While it provides immediate execution of bytecode, it is generally slower than the JIT compiler because it doesn’t perform optimizations.

**b. JIT (Just-In-Time) Compiler:**

* **Role:** Compiles bytecode into native machine code at runtime to improve execution speed.
* **Scope:** Operates during program execution.
* **Details:** Translates frequently executed bytecode into native code, which is cached and reused. This reduces the overhead of interpretation and improves performance.

**c. Garbage Collector:**

* **Role:** Manages memory by automatically reclaiming memory occupied by objects that are no longer reachable.
* **Scope:** Runs in the background to free up memory.
* **Details:** Various garbage collection algorithms (e.g., Mark and Sweep, Generational Garbage Collection) are used to optimize memory management.

### ****4. Program Termination:****

**Role:**

* Manages the shutdown of the JVM once the program execution is complete.
* Involves performing cleanup operations such as finalizing garbage collection, releasing resources, and terminating threads.

**Details:**

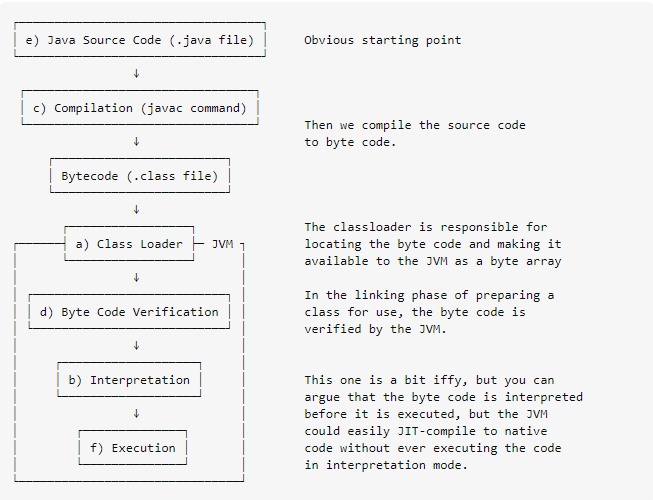
* **Shutdown Hooks:** The JVM provides a mechanism for executing code before the JVM terminates (e.g., releasing resources, saving state).
* **Resource Management:** Ensures all resources (e.g., file handles, network connections) are properly closed.

### ****Summary of JVM Execution Flow:****

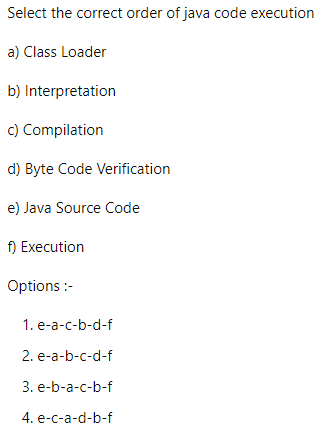
1. **Class Loading:**
   * Class files are loaded into memory by the class loader.
   * The class loader handles the loading, linking, and initialization of classes.
2. **Memory Management:**
   * The JVM allocates memory for classes, objects, and thread-specific data.
   * The garbage collector manages the heap area and reclaims memory.
3. **Execution:**
   * Bytecode is executed either by the interpreter or JIT compiler.
   * The execution engine processes instructions and manages method calls and local variables.
4. **Program Termination:**
   * The JVM performs final cleanup operations and releases resources.
   * It shuts down gracefully and ensures all threads are terminated.

### ****Summary****

* **Method Area:** Stores static variables, static blocks, static methods, and class metadata.
* **Heap Area:** Stores objects and instance variables.
* **Stack Area:** Stores local variables and method call information.
* **PC Register:** Tracks the current instruction.
* **Native Method Stack:** Handles native method calls.
* **String Pool:** Special area in the method area for storing string literals.



Q. Select correct option below:



Note: Lifecycle and execution flow of the java pram link for reference below:

<https://www.cesarsotovalero.net/blog/how-the-jvm-executes-java-code.html>

<https://medium.com/@fullstacktips/understanding-the-order-of-code-execution-in-java-classes-8f5f865d0ccb>

<https://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html>

<https://www.geeksforgeeks.org/compilation-execution-java-program/>

**Q. What are the main features of oops and explain it?**

There are six main features of oops in java

1. Class
2. Object
3. Inheritance
4. Polymorphism
5. Encapsulation
6. Abstraction

**Class→** class in a **group of objects** and class is **not a real world entity** it is just **blueprint or template** and **class does not occupy memory.**

**Ex:-Animal,vehicle**

**Object→** object is an **instance of a class** and object is a **real world entity** and **object occupies memory** and every object consists of **identity**(name of object), **attribute**(color, age, breed) and **behavior**(run, eat, walk, sleep)**.**

**Ex:**-**inside the animal class dog,cat,rat is an object name which identity,attribute and behavior.**

**Inheritance→** inheritance is a process in which a **child class acquires the properties of the parent class** called inheritance and it has three types.

**Ex:-son acquired the properties from father.**

1. Single level
2. Multi level
3. Hierarchical
4. Hybride(not supported by java)
5. Multiple(not supported by java)

**Polymorphism→** The word **polymorphism means having many forms** called polymorphism in another way polymorphism in which one object has many forms called polymorphism and there are two types of polymorphism.

1. Compile time/static polymorphism**(achieved by method overloading)**
2. Runtime/Dynamic polymorphism**(achieved by method overriding)**

**Encapsulation→** wrapping a **data member and member function** together in a **single unit** called encapsulation.We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use **setter and getter methods to set and get the data** in it. **java bean class is an example of a fully encapsulated class**.

**Abstraction→ is** a process of **hiding the implementation details** and showing only **functionally** to the user called abstraction and there are two ways to achieve abstraction in java.

1. **Abstract class**(0-100 percent abstraction we can achieve).
2. **Interface**(100 percent abstraction we can achieve).

Ex:-atm machine, whatsapp how they are working we don't know we are just using it like several functions and services are available.

**Q. What is constructor and its types?**

* constructor is a block (similar to method) having the same name as that of class name called constructor.
* constructor does not have any return type even void.
* constructor execute automatically when we create an object.
* Only four modifiers are applicable for constructors: public,private,protected and default.

**There are 3 type in java**

1. Default constructor
2. User define constructor
3. Parameterized constructor

**Note:** Uses of the constructor to initialization of an object.

**Example:**

In Java, there are three types of constructors:

1. **Default Constructor**:
   * Provided by the compiler if no constructors are explicitly defined in the class.
   * Has no parameters.
2. **No-Argument Constructor**:
   * Defined by the programmer.
   * Does not take any parameters.
3. **Parameterized Constructor**:
   * Defined by the programmer.
   * Takes one or more parameters to initialize the object with specific values.

**Example Program with All Types of Constructors**

public class Car {

// Attributes

private String brand;

private String model;

private int year;

// 1. Default Constructor (implicitly provided by the compiler)

// If no constructor is defined, the compiler provides a default constructor.

// This constructor does nothing but allows creating objects without initialization.

// 2. No-Argument Constructor (explicitly defined)

public Car() {

this.brand = "Unknown";

this.model = "Unknown";

this.year = 0;

System.out.println("No-Argument Constructor called");

}

// 3. Parameterized Constructor

public Car(String brand, String model, int year) {

this.brand = brand;

this.model = model;

this.year = year;

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

}

// Method to display car details

public void displayDetails() {

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

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

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

}

public static void main(String[] args) {

// Using the Default Constructor (only available if no other constructors are defined)

// Car defaultCar = new Car(); // Uncommenting this line will cause a compilation error if no constructors are defined.

// Using the No-Argument Constructor

Car car1 = new Car();

car1.displayDetails(); // Output: Unknown, Unknown, 0

System.out.println();

// Using the Parameterized Constructor

Car car2 = new Car("Toyota", "Corolla", 2020);

car2.displayDetails(); // Output: Toyota, Corolla, 2020

}

}

**Explanation:**

1. **Default Constructor**:
   * The default constructor is provided by the compiler if no other constructors are defined.
   * If you define any constructor (either no-argument or parameterized), the default constructor is no longer provided by the compiler.
2. **No-Argument Constructor**:
   * The no-argument constructor is explicitly defined by the programmer.
   * It initializes the attributes with default values.
3. **Parameterized Constructor**:
   * The parameterized constructor takes arguments to initialize the attributes with specific values.
   * It allows creating objects with customized initial values.

**Output:**

When you run the program, the output will be:

No-Argument Constructor called

Brand: Unknown

Model: Unknown

Year: 0

Parameterized Constructor called

Brand: Toyota

Model: Corolla

Year: 2020

**Summary:**

* The **No-Argument Constructor** is used when you want to create objects with default values.
* The **Parameterized Constructor** allows creating objects with specific initial values.
* The **Default Constructor** is automatically provided by the compiler when no other constructors are defined in the class. However, if any constructor is defined, the default constructor must be explicitly declared if needed.

**Q. Example of Class, object and Constructor in java?**

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

A class is a blueprint for creating objects. It defines the attributes (properties) and behaviors (methods) of the objects.

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

An object is an instance of a class. It is created using the new keyword and has its own state (defined by the attributes).

#### 3. **Constructor**:

A constructor is a special method that is called when an object is instantiated. It is used to initialize the object’s attributes.

### Example Code:

// Defining a class named Car

class Car {

// Attributes (properties)

String brand;

String model;

int year;

// Constructor: used to initialize the attributes

Car(String brand, String model, int year) {

this.brand = brand;

this.model = model;

this.year = year;

}

// Method to display car details

void displayDetails() {

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

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

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

}

}

public class Main {

public static void main(String[] args) {

// Creating an object of the Car class using the constructor

Car myCar = new Car("Toyota", "Corolla", 2020);

// Calling the method to display car details

myCar.displayDetails();

}

}

### Explanation:

1. **Class Definition**:
   * The Car class defines three attributes: brand, model, and year.
   * It also has a constructor Car(String brand, String model, int year) that initializes these attributes when a Car object is created.
2. **Object Creation**:
   * In the Main class, an object myCar of the Car class is created using the new keyword and the constructor: new Car("Toyota", "Corolla", 2020).
   * This initializes the myCar object with the brand "Toyota", model "Corolla", and year 2020.
3. **Method Call**:
   * The displayDetails() method of the myCar object is called to print the car's details to the console.

### Output:

Brand: Toyota

Model: Corolla

Year: 2020

**Q. Elements of the class below or class can contains?**

In Java, a class can contain several elements that define its structure and behavior. These elements can be grouped into three main categories: attributes, methods, and additional components like constructors, blocks, and nested classes.

**Elements of a Class:**

1. **Attributes (Fields/Properties/Variables)**:
   * These are variables that hold the state or data of an object.
   * **Example**:

String brand;

int year;

1. **Methods (Functions/Behaviors)**:
   * Methods define the actions or behaviors that objects of the class can perform.
   * **Example**:

void startEngine() {

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

}

1. **Constructors**:
   * Constructors are special methods used to initialize objects. They have the same name as the class and do not have a return type.
   * **Example**:

Car(String brand, int year) {

this.brand = brand;

this.year = year;

}

1. **Blocks**:
   * These include instance initializer blocks, static initializer blocks, and synchronized blocks.
   * **Example**:

// Instance initializer block

{

System.out.println("Instance initializer block");

}

// Static initializer block

static {

System.out.println("Static initializer block");

}

1. **Nested Classes**:
   * A class can contain other classes, which are called nested or inner classes. These can be static or non-static.
   * **Example**:

class Engine {

// Nested class

}

1. **Access Modifiers**:
   * These are keywords that set the visibility and accessibility of the class, its methods, and its attributes. The common access modifiers are private, protected, public, and the default (package-private).
   * **Example**:

private String brand;

public void startEngine() {}

1. **Static Members**:
   * Static members belong to the class rather than any specific object. They include static variables and static methods.
   * **Example**:

static int numberOfCars;

static void displayTotalCars() {

System.out.println("Total cars: " + numberOfCars);

}

1. **Final Members**:
   * The final keyword can be used to create constants or to prevent methods from being overridden and classes from being subclassed.
   * **Example**:

final String VIN; // Vehicle Identification Number

1. **Interfaces and Abstract Classes**:
   * A class can implement interfaces or extend an abstract class, providing specific implementations for abstract methods.
   * **Example**:

interface Drivable {

void drive();

}

abstract class Vehicle {

abstract void fuel();

}

**Example:**

public class Car {

// Attributes (Fields)

private String brand;

private String model;

private int year;

private static int numberOfCars;

// Constructor

public Car(String brand, String model, int year) {

this.brand = brand;

this.model = model;

this.year = year;

numberOfCars++;

}

// Method to start the engine

public void startEngine() {

System.out.println("Engine started for " + brand + " " + model);

}

// Static Method to display total number of cars

public static void displayTotalCars() {

System.out.println("Total cars: " + numberOfCars);

}

// Nested Class

class Engine {

void run() {

System.out.println("Engine running for " + brand + " " + model);

}

}

// Static Block

static {

numberOfCars = 0;

}

// Instance Initializer Block

{

System.out.println("Car object created");

}

// Main method to execute the program

public static void main(String[] args) {

// Creating an object of Car class

Car myCar = new Car("Toyota", "Corolla", 2020);

// Calling method to start the engine

myCar.startEngine();

// Calling static method to display total number of cars

Car.displayTotalCars();

// Creating and using an instance of the nested class Engine

Car.Engine myEngine = myCar.new Engine();

myEngine.run();

}

}

**Q. Object initialization using reference variable, method and constructor in java?**

Here’s a detailed explanation of how to initialize an object in Java using different approaches: by reference variable, by method, and by constructor. I'll provide code examples for each approach.

**1. Object Initialization Using a Reference Variable**

In this approach, you create an object using a constructor and then use the reference variable to initialize or modify the object's attributes.

public class Car {

private String brand;

private String model;

private int year;

// Constructor

public Car(String brand, String model, int year) {

this.brand = brand;

this.model = model;

this.year = year;

}

// Method to display car details

public void displayDetails() {

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

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

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

}

public static void main(String[] args) {

// Object creation and initialization using constructor

Car myCar = new Car("Toyota", "Corolla", 2020);

// Display car details

myCar.displayDetails();

}

}

**2. Object Initialization Using a Method**

In this approach, you first create an object using a constructor and then use a method to initialize or set its attributes.

public class Car {

private String brand;

private String model;

private int year;

// No-Argument Constructor

public Car() {}

// Method to initialize car attributes

public void initialize(String brand, String model, int year) {

this.brand = brand;

this.model = model;

this.year = year;

}

// Method to display car details

public void displayDetails() {

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

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

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

}

public static void main(String[] args) {

// Object creation using no-argument constructor

Car myCar = new Car();

// Initialize object attributes using method

myCar.initialize("Honda", "Civic", 2019);

// Display car details

myCar.displayDetails();

}

}

**3. Object Initialization Using a Constructor**

Here’s how you can initialize an object directly through different types of constructors: no-argument and parameterized constructors.

public class Car {

private String brand;

private String model;

private int year;

// No-Argument Constructor

public Car() {

this.brand = "Unknown";

this.model = "Unknown";

this.year = 0;

}

// Parameterized Constructor

public Car(String brand, String model, int year) {

this.brand = brand;

this.model = model;

this.year = year;

}

// Method to display car details

public void displayDetails() {

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

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

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

}

public static void main(String[] args) {

// Object initialization using no-argument constructor

Car car1 = new Car();

car1.displayDetails(); // Output: Unknown, Unknown, 0

// Object initialization using parameterized constructor

Car car2 = new Car("Ford", "Mustang", 2021);

car2.displayDetails(); // Output: Ford, Mustang, 2021

}

}

**Summary**

1. **Using a Reference Variable**:
   * Create the object and initialize it using the constructor. Attributes are set at object creation time.
2. **Using a Method**:
   * Create the object and initialize its attributes through a method call after object creation.
3. **Using a Constructor**:
   * Initialize the object with either a no-argument constructor (for default values) or a parameterized constructor (for specific values).

**Q. Inheritance code below**

**Single-Level Inheritance**

In single-level inheritance, a class (subclass) inherits from another class (superclass).

**Example**:

// Superclass

class Animal {

void eat() {

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

}

}

// Subclass inheriting from Animal

class Dog extends Animal {

void bark() {

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

}

}

public class SingleLevelInheritanceDemo {

public static void main(String[] args) {

Dog myDog = new Dog();

myDog.eat(); // Method from Animal class

myDog.bark(); // Method from Dog class

}

}

**Explanation**:

* Dog class inherits the eat() method from the Animal class.
* Dog class also has its own method bark().

**2. Multi-Level Inheritance**

In multi-level inheritance, a class (subclass) is derived from another subclass, forming a chain of inheritance.

**Example**:

// Base class

class Vehicle {

void start() {

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

}

}

// Intermediate class

class Car extends Vehicle {

void drive() {

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

}

}

// Derived class

class SportsCar extends Car {

void turboBoost() {

System.out.println("Sports car turbo boost!");

}

}

public class MultiLevelInheritanceDemo {

public static void main(String[] args) {

SportsCar mySportsCar = new SportsCar();

mySportsCar.start(); // Method from Vehicle class

mySportsCar.drive(); // Method from Car class

mySportsCar.turboBoost(); // Method from SportsCar class

}

}

**Explanation**:

* SportsCar inherits from Car, which inherits from Vehicle.
* SportsCar can access methods from both Car and Vehicle, along with its own method turboBoost().

**3. Hierarchical Inheritance**

In hierarchical inheritance, multiple subclasses inherit from a single superclass.

**Example**:

// Superclass

class Animal {

void eat() {

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

}

}

// Subclass 1

class Dog extends Animal {

void bark() {

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

}

}

// Subclass 2

class Cat extends Animal {

void meow() {

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

}

}

public class HierarchicalInheritanceDemo {

public static void main(String[] args) {

Dog myDog = new Dog();

Cat myCat = new Cat();

myDog.eat(); // Method from Animal class

myDog.bark(); // Method from Dog class

myCat.eat(); // Method from Animal class

myCat.meow(); // Method from Cat class

}

}

**Explanation**:

* Both Dog and Cat inherit from the Animal class.
* Each subclass has its own unique methods but also shares the eat() method from the Animal class.

**Summary**

* **Single-Level Inheritance**: A subclass inherits directly from a single superclass.
* **Multi-Level Inheritance**: A chain of inheritance where a class inherits from another subclass.
* **Hierarchical Inheritance**: Multiple subclasses inherit from a single superclass

**Q.Polymorphism program below;**

**1. Compile-time Polymorphism (Method Overloading)**

Method overloading allows a class to have more than one method with the same name but different parameters (different type or number of parameters).

**Example**:

class MathOperations {

// Method to add two integers

int add(int a, int b) {

return a + b;

}

// Method to add three integers

int add(int a, int b, int c) {

return a + b + c;

}

// Method to add two doubles

double add(double a, double b) {

return a + b;

}

}

public class CompileTimePolymorphismDemo {

public static void main(String[] args) {

MathOperations math = new MathOperations();

// Calling overloaded methods

System.out.println("Sum of 10 and 20: " + math.add(10, 20));

System.out.println("Sum of 10, 20, and 30: " + math.add(10, 20, 30));

System.out.println("Sum of 10.5 and 20.5: " + math.add(10.5, 20.5));

}

}

**Explanation**:

* The add method is overloaded with different parameter lists.
* The appropriate method is selected at compile-time based on the arguments passed.

**2. Run-time Polymorphism (Method Overriding)**

Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. The method in the subclass should have the same name, return type, and parameters as the method in the superclass.

**Example**:

class Main {

public void m1(int a, int b) {

int c = a + b;

System.out.println("Adding two numbers in Main: " + c);

}

}

class B extends Main {

@Override

public void m1(int a, int b) {

super.m1(1, 2); // Call the m1 method from Main with fixed arguments

int c = a - b;

System.out.println("Subtracting two numbers in B class: " + c);

}

}

class C extends B {

@Override

public void m1(int a, int b) {

super.m1(1, 2); // Call the m1 method from B with fixed arguments

int c = a \* b;

System.out.println("Multiplying two numbers in C class: " + c);

}

}

public class Test {

public static void main(String[] args) {

C c = new C();

c.m1(1, 2); // This will call the m1 method in C

super.m1(1,3); //not valid

// You cannot use super directly in the static main method.

// Instead, use instance methods to call super methods.

}

}

**Explanation**:

* The sound method in the Animal class is overridden in both Dog and Cat subclasses.
* The method that gets executed is determined at runtime based on the object type (Dog or Cat) that myAnimal refers to.

**Summary**

* **Compile-time Polymorphism (Method Overloading)**: Achieved by defining multiple methods with the same name but different parameter lists in the same class.
* **Run-time Polymorphism (Method Overriding)**: Achieved by redefining a method in a subclass with the same signature as in the superclass, allowing dynamic method dispatch at runtime.

Polymorphism is a core concept in object-oriented programming that enhances flexibility and maintainability in your code by allowing objects to be manipulated through a common interface.

**Q.Encapsulation program below:**

1. **Encapsulation**: Encapsulation is the practice of hiding the internal state and requiring all interactions to be performed through methods. This ensures that the internal representation of the object is protected from outside interference and misuse.
2. **Classes and Methods**: The code contains three classes: Ecapsulation1, Ecapsulation, and A. Each class has its own purpose and demonstrates different aspects of encapsulation.

### Detailed Explanation

#### 1. **Class** Ecapsulation1

This is the main class containing the main method, which serves as the entry point of the application. It demonstrates how to use the Ecapsulation class and the A class.

public class Ecapsulation1 {

public static void main(String[] args) {

// Create an instance of Ecapsulation

Ecapsulation e = new Ecapsulation();

e.setId(102);

e.setCity("Bangalore");

e.setCollege("CMRIT");

// Print the values using getter methods

System.out.println(e.getId());

System.out.println(e.getCity());

System.out.println(e.getCollege());

System.out.println("---------------------------------");

// Create an instance of class A and call its method

A a = new A();

a.initialize();

}

}

* **Creating an Instance of Ecapsulation**: An object e of the class Ecapsulation is created. Methods setId(), setCity(), and setCollege() are called to set values.
* **Printing Values**: The getter methods getId(), getCity(), and getCollege() are used to retrieve and print the values.
* **Creating an Instance of A**: An object a of the class A is created, and its initialize() method is called.

#### 2. **Class** Ecapsulation

This class demonstrates encapsulation by using private fields and public getter and setter methods to access and modify the private fields.

class Ecapsulation {

private int id;

private String city;

private String college;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

public String getCollege() {

return college;

}

public void setCollege(String college) {

this.college = college;

}

}

* **Private Fields**: The fields id, city, and college are private, which means they cannot be accessed directly from outside the class.
* **Public Getter and Setter Methods**: These methods provide controlled access to the private fields. setId(), setCity(), and setCollege() are used to set values, while getId(), getCity(), and getCollege() are used to retrieve them.

#### 3. **Class** A

This class has a method initialize() that demonstrates how to use the Ecapsulation class to create and manipulate objects.

class A {

void initialize() {

// Create an instance of Ecapsulation

Ecapsulation e = new Ecapsulation();

e.setId(101);

e.setCity("Delhi");

e.setCollege("CMRIT");

// Print the values using getter methods

System.out.println(e.getId());

System.out.println(e.getCity());

System.out.println(e.getCollege());

}

}

* **Creating an Instance of Ecapsulation**: An object e of the class Ecapsulation is created within the initialize() method.
* **Setting and Printing Values**: The object’s fields are set using setter methods, and the values are printed using getter methods.

### Summary

* **Encapsulation**: The Ecapsulation class uses encapsulation by keeping its data private and providing public methods to access and modify that data.
* **Object Creation**: Objects of Ecapsulation and A are created in the main method and initialize() method, respectively.
* **Method Usage**: Methods in both Ecapsulation and A demonstrate how encapsulated objects can be manipulated and accessed.

This design ensures that the internal state of the Ecapsulation class is protected and only modified through well-defined methods, adhering to the principle of encapsulation in object-oriented programming.

**Q. Abstraction Program with notes:**

**Two way to achieve abstraction in java below is written**

### 1. ****Abstract Classes****

An abstract class is a class that cannot be instantiated on its own and is meant to be subclassed. It may contain abstract methods (methods without a body) that must be implemented by subclasses, as well as non-abstract methods (methods with a body).

### ****2. Interfaces****

An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces are used to achieve abstraction and multiple inheritance.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| abstract class must be declare with abstract keyword | interface must be declared with interface keyword. |
| abstract classes can have an abstract and non-abstract method. | interface can have only abstract methods. |
| we can not create objects of abstract class. | we can not create objects of interface. |
| abstract class does not support multiple inheritance. | interface supports multiple inheritance. |
| abstract class used to achieve (0-100) percent abstraction in java. | interface  used to achieve (100) percent abstraction in java. |
| abstract class can have (1)abstract and (2)non-abstract method as well as (3)simple block, (4)static block, (5)constructor, (6)static method, (7)final method and (8)static and (9)final variable. | interface can have **static method,Default method since java 8**,by **default every variable in interface is public,static and fina**l and **every method is public and abstract** we can not use  block and constructor inside the interface. |

**Note:-we can not use constructor and block inside interface.**

**Abstract class program below:**

**public** **abstract** **class** AbstractClass {

**static** **int** *x* = 10; //static variable

**final** **int** y = 20; //final variable

**abstract** **void** run1(); //abstract method

**void** run2() { //Non-abstract method

System.***out***.println("non abstract method..");

}

AbstractClass() { //constructor

System.***out***.println("this is constructor...");

}

{ //block

System.***out***.println("this is block");

}

**static** { //static block

System.***out***.println("this is static block");

}

**static** **void** static\_method() { //static method

System.***out***.println("this is static method");

}

**final** **void** final\_method() { //final method

System.***out***.println("this is final method");

}

}

**class** Amarjeet1 **extends** AbstractClass{

@Override

**void** run1() {

System.***out***.println("this is abstract method");

}

**public** **static** **void** main(String[] args) {

Amarjeet1 am=**new** Amarjeet1();

System.***out***.println(am.y);

System.***out***.println(AbstractClass.*x*);

am.final\_method();

am.run2();

am.run1();

AbstractClass.*static\_method*();

}

}

//abstract class can have abstract method

//abstract class can have non-abstract method

//abstract class can have constructor

//abstract class can have static method

//abstract class can have final method

//abstract class can have concreate method

**Interface program below:**

**interface** InterfaceDemo {

**abstract** **void** m1();

**abstract** **void** m2();

**public** **void** m2();

**default** **void** default\_method() {

System.***out***.println("Default method");

}

**static** **void** static\_method() {

System.***out***.println("This is static method...");

}

}

**class** B **implements** InterfaceDemo {

**public** **void** m1() {

System.***out***.println("calling m1() of interface...");

}

**public** **void** m2() {

System.***out***.println("calling m1() of interface...");

}

**public** **static** **void** main(String[] args) {

B b = **new** B();

b.m1();

b.m2();

InterfaceDemo.*static\_method*();

}

**public** **void** m2() {

System.***out***.println("calling m2() of interface...");

}

}

**Rules in java for oops conccepts:**

In Object-Oriented Programming (OOP), certain rules and principles govern the design and implementation of object-oriented systems. These rules ensure that the code adheres to the core principles of OOP and is maintainable, scalable, and reusable. Here’s a list of key rules and regulations in OOP concepts:

**1. Encapsulation**

* **Private Members**: Data members (fields) should be private to prevent direct access from outside the class. This hides the internal state and allows modification only through methods.
* **Public Methods**: Provide public getter and setter methods to access and modify the private data fields.

**2. Abstraction**

* **Abstract Classes**: Cannot be instantiated and may contain abstract methods (methods without implementation) that must be implemented by subclasses.
* **Interfaces**: Define methods that a class must implement. Interfaces support multiple inheritance and can include default methods and static methods.

**3. Inheritance**

* **Single Inheritance**: A class can inherit from only one superclass, which is the default behavior in languages like Java.
* **Multiple Inheritance**: Java does not support multiple inheritance through classes (a class cannot inherit from more than one class) but supports it through interfaces.
* **Overriding**: Subclasses can provide a specific implementation of methods that are already defined in their superclass using the @Override annotation.

**4. Polymorphism**

* **Method Overloading**: Multiple methods in the same class with the same name but different parameters. This is compile-time polymorphism.
* **Method Overriding**: A subclass provides a specific implementation of a method that is already defined in its superclass. This is runtime polymorphism.
* **Dynamic Method Dispatch**: The method that gets executed is determined at runtime based on the object’s actual type.

**5.Class Rules**

* **Definition**: A class is a blueprint for creating objects. It defines the properties (fields) and behaviors (methods) that the objects created from the class will have.
* **Access Modifiers**:
  + public: The class can be accessed from any other class.
  + protected: The class can be accessed by subclasses and classes in the same package.
  + default (no modifier): The class is accessible only within its own package.
  + private: Not allowed for top-level classes (only for inner classes).
* **Naming Conventions**:
  + Class names should be in PascalCase (e.g., StudentInfo).
  + Use meaningful names that reflect the purpose of the class.
* **Fields**:
  + Typically private to encapsulate data.
  + Use getters and setters to access and modify private fields.
* **Methods**:
  + Define the behaviors of the class.
  + Use appropriate access modifiers (public, protected, private).
* **Static Members**:
  + Static fields and methods belong to the class itself rather than instances of the class.
* **Inheritance**:
  + A class can extend only one other class (single inheritance).
  + Use extends keyword to inherit from a superclass.

**6. Object Rules**

* **Creation**:
  + An object is created using the new keyword followed by a constructor call.
  + Example: Student s = new Student();
* **Access**:
  + Access the object's fields and methods using the dot operator (e.g., s.getName()).
* **Initialization**:
  + An object should be properly initialized before using its fields and methods.

**7. Constructor Rules**

* **Definition**:
  + A constructor is a special method that is called when an object is instantiated. It initializes the object.
  + Constructors have the same name as the class and do not have a return type.
* **Types of Constructors**:
  + **Default Constructor**: Provided by the compiler if no constructor is explicitly defined. It initializes all fields to default values.
  + **Parameterized Constructor**: Allows initializing fields with specific values when an object is created. Defined with parameters.
  + **Copy Constructor**: (In Java, not explicitly supported) Can be simulated using a constructor that takes an instance of the same class as a parameter.
* **Rules**:
  + **No Return Type**: Constructors do not have a return type, not even void.
  + **Overloading**: Multiple constructors can be defined in a class with different parameters (constructor overloading).
  + **Chaining**: Constructors can call other constructors in the same class using this() or the superclass constructor using super().
  + **Initialization Block**: Can use initialization blocks to initialize common setup code. Initialization blocks are executed before constructors.