**1. What is Java?**

**Java** is a high-level, object-oriented programming language that was designed to be portable, meaning it can run on any device or operating system that has the Java Virtual Machine (JVM) installed. Java programs are compiled into bytecode, which the JVM interprets and runs.

* **Example:**

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!"); // Prints Hello, World! to the console

}

}

In this example, the HelloWorld class has a main method, which is the entry point for any Java application. It prints "Hello, World!" to the console.

**Java Editions:**

Java has multiple editions, each designed to cater to different types of application development. These editions provide various libraries, APIs, and tools to help developers build specific types of applications. Here's a breakdown of the main **Java Editions**:

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

Java SE is the core Java platform and provides the foundational libraries and APIs for developing general-purpose applications. It includes all the basic features of the Java language and standard libraries.

* **Key Features:**
  + Core Java libraries (e.g., java.lang, java.util, java.io)
  + APIs for networking, database interaction, multithreading, and collections
  + Java Virtual Machine (JVM) for running Java applications
  + Tools like the Java compiler (javac), java command for running applications, and the Java Debugger (jdb)
  + Common libraries for file I/O, security, and networking
* **Applications Built with Java SE:**
  + Desktop applications
  + Standalone applications
  + Command-line utilities
  + Core back-end logic for enterprise applications
* **Version Example:** Java SE 8, Java SE 11 (LTS), Java SE 17 (LTS)

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

Java EE (now known as **Jakarta EE**) is an extension of Java SE, and it's designed for building large-scale, distributed, and multi-tier enterprise applications. It includes additional libraries and specifications for building web applications and enterprise-grade systems.

* **Key Features:**
  + Servlets and JSP (Java Server Pages) for web applications
  + EJB (Enterprise JavaBeans) for building scalable, distributed applications
  + JPA (Java Persistence API) for database interaction and ORM (Object Relational Mapping)
  + JMS (Java Message Service) for messaging between systems
  + JAX-RS and JAX-WS for building RESTful and SOAP-based web services
  + Dependency injection, transactions, and security services
* **Applications Built with Java EE:**
  + Large enterprise web applications
  + E-commerce platforms
  + Cloud-based and microservice-based applications
  + RESTful services
* **Version Example:** Jakarta EE 8, Java EE 7, Jakarta EE 9

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

Java ME is designed for embedded systems, mobile devices, and other small devices like sensors and smart appliances. It provides a smaller subset of the Java SE libraries, tailored for devices with limited resources.

* **Key Features:**
  + Optimized for small devices with limited memory and processing power
  + APIs specific to mobile and embedded development, including user interface, networking, and storage capabilities for constrained environments
  + CLDC (Connected Limited Device Configuration) and CDC (Connected Device Configuration) profiles
* **Applications Built with Java ME:**
  + Mobile applications (earlier versions of mobile platforms)
  + Embedded systems like IoT (Internet of Things) devices
  + Consumer electronics (smartwatches, MP3 players, etc.)
* **Version Example:** Java ME 8, Java ME 8.3

**Java Versions Overviews:**

### **1. Java 1.0 (January 1996)**

* **Key Features**:
  + First official release of Java.
  + Introduced the concept of Write Once, Run Anywhere (WORA).
  + Basic object-oriented features (inheritance, encapsulation, etc.).
  + Supported primitive types, objects, and references.
  + **AWT** (Abstract Window Toolkit) for GUI.
  + Applets for embedding Java programs in web browsers.
  + **java.io** for I/O operations.
  + Multi-threading, networking, and memory management with automatic garbage collection.
  + Java bytecode and the Java Virtual Machine (JVM) concept.

### **2. Java 1.1 (February 1997)**

* **Key Features**:
  + **Inner classes** introduced (local, anonymous, static, and non-static inner classes).
  + Major revision of the **event handling model**.
  + **JDBC (Java Database Connectivity)**: Basic database interaction API.
  + **RMI (Remote Method Invocation)**: Allows invoking methods on remote objects.
  + Reflection API: Enables runtime retrieval of information about classes and methods.
  + **JavaBeans**: Reusable software components (getter/setter methods).
  + Internationalization (support for different languages and regions).
  + CORBA support for distributed computing.
  + **Object serialization** for object persistence and transmission.

### **3. Java 1.2 (December 1998) – "Java 2"**

* **Key Features**:
  + Rebranding of the platform as Java 2.
  + **Swing** introduced: A new GUI toolkit replacing AWT components.
  + **Collections Framework** introduced: Data structures such as List, Set, Map, etc.
  + **Just-In-Time (JIT) compiler** improvements for better performance.
  + **Java Plug-in** for embedding Java applets into web browsers.
  + **Security enhancements**: Signed applets, permissions model, and AccessControl.
  + **Java Foundation Classes (JFC)**, including Swing, accessibility, drag-and-drop, and more.

### **4. Java 1.3 (May 2000)**

* **Key Features**:
  + **HotSpot JVM** becomes the default: Optimizes performance through JIT compilation.
  + **Java Sound API** for sound and audio manipulation.
  + **RMI over IIOP**: RMI interoperates with CORBA (Common Object Request Broker Architecture).
  + Enhancements in Java AWT (lightweight components).
  + Improvements in the **JDBC API** and **JNDI** (Java Naming and Directory Interface).

### **5. Java 1.4 (February 2002)**

* **Key Features**:
  + **Java NIO (New I/O)**: Non-blocking I/O, buffers, selectors, and channels for scalable I/O operations.
  + **Assertions** introduced to test assumptions in the code.
  + **Chained exceptions**: Allows associating exceptions for better debugging.
  + **Logging API**: Provides logging facilities within the Java SE platform.
  + **Regular Expressions** support via java.util.regex package.
  + XML processing (JAXP) introduced.
  + Security enhancements: Java Cryptography Architecture (JCA), JSSE, and others.
  + Performance improvements in garbage collection and JVM execution.

### **6. Java 5 (September 2004) – "Java 1.5"**

* **Key Features**:
  + **Generics**: Strongly typed collections (e.g., List<String>).
  + **Enhanced for-loop**: Simplified looping through collections and arrays.
  + **Autoboxing and unboxing**: Automatic conversion between primitive types and their wrapper objects.
  + **Enums**: Introduced as a special class type.
  + **Varargs**: Allows passing variable-length arguments.
  + **Annotations**: Metadata for classes, methods, fields, etc., (@Override, @Deprecated, @SuppressWarnings).
  + **Concurrency utilities**: Java 5 introduced the java.util.concurrent package with features like Executor, Future, and Semaphore.
  + **Static imports**: To use static members of classes without prefixing class names.
  + JVM improvements: Better performance and memory management.

### **7. Java 6 (December 2006)**

* **Key Features**:
  + **Scripting language support (JSR 223)**: Integrates with scripting languages like JavaScript.
  + **JAX-WS**: Standard for web services development.
  + **Pluggable annotations**: Allows custom annotations processing using APT (Annotation Processing Tool).
  + Improvements to **Swing** for GUI development.
  + **Compiler API**: Programmatically compile Java code using the javax.tools package.
  + **Java Compiler (javac)** improvements.
  + **Monitoring and Management**: Enhanced JVM monitoring, management using JMX (Java Management Extensions).
  + Performance optimizations in the **JIT compiler** and JVM.

### **8. Java 7 (July 2011)**

* **Key Features**:
  + **Try-with-resources**: Automatic resource management in try blocks.
  + **Diamond operator** (<>): Simplifies code involving generics.
  + **Switch with Strings**: Allows switch statement to accept String types.
  + **Fork/Join framework**: Simplifies parallelism in applications using multiple processors.
  + **NIO.2 (Non-blocking I/O 2)**: Advanced file operations, symbolic links, and new Path API.
  + **Multicatch**: Handle multiple exceptions in a single catch block.
  + Binary literals and underscores in numeric literals for better readability.
  + **G1 (Garbage First) garbage collector**: A low-pause garbage collector.

### **9. Java 8 (March 2014)**

* **Key Features**:
  + **Lambda expressions**: Enables functional programming in Java.
  + **Stream API**: Allows functional-style operations on collections.
  + **Default methods in interfaces**: Provides method implementations in interfaces.
  + **Optional class**: Helps avoid NullPointerException by wrapping nullable values.
  + **New Date and Time API** (java.time): Modern date/time management classes like LocalDate, LocalTime, LocalDateTime, etc.
  + **Method references**: Shorter lambda expressions by referring to methods (ClassName::methodName).
  + **Functional interfaces**: Interfaces with a single abstract method, e.g., Runnable, Callable, Supplier.
  + **Nashorn JavaScript engine**: Replaces the older Rhino engine.
  + **Repeating annotations**: Multiple annotations of the same type can be applied to a declaration.

### **10. Java 9 (September 2017)**

* **Key Features**:
  + **Modular System (Project Jigsaw)**: Breaks the JDK into modules, enabling better scaling and performance.
  + **JShell**: A REPL (Read-Eval-Print Loop) for experimenting with Java snippets.
  + **Private methods in interfaces**: Allows private methods inside interfaces.
  + **Factory methods for collections**: List.of(), Set.of(), Map.of() for creating immutable collections.
  + **HTTP/2 Client API** (incubating): Supports HTTP/2 protocol.
  + Improved performance and memory handling for garbage collection.
  + Multi-version JARs: JARs that can support multiple Java versions.
  + Process API improvements for controlling and managing OS processes.

### **11. Java 10 (March 2018)**

* **Key Features**:
  + **Local-Variable Type Inference (var)**: Java introduces var to infer types of local variables.
  + **Garbage Collector Interface**: Improves modularity of garbage collectors.
  + **Application Class-Data Sharing**: Reduces startup time by sharing class data between JVMs.
  + **Thread-Local Handshakes**: Enables JVM to stop individual threads and not stop all threads for a safepoint.
  + **Experimental JIT Compiler (Graal)**: Available as an experimental feature.

### **12. Java 11 (September 2018)** – **LTS** release

* **Key Features**:
  + **HTTP Client API** (standardized): Supports HTTP/2 and WebSocket.
  + **New String Methods**: strip(), repeat(), lines(), and isBlank().
  + **Nest-Based Access Control**: Improves access between nested classes.
  + **Lambda Parameter Syntax**: Supports local-variable syntax for lambda parameters.
  + Removal of **Java EE** and **CORBA** modules.
  + **Single File Source Code**: Run Java files directly from the command line without compilation (java HelloWorld.java).

### **13. Java 12 (March 2019)**

* **Key Features**:
  + **Switch Expressions (preview)**: Allows switch to return values and simplifies syntax.
  + **Shenandoah garbage collector**: A low-pause garbage collector.
  + **JVM Constants API**: Simplifies JVM support for dynamic languages.
  + **Microbenchmark Suite**: Tool for writing microbenchmarks for Java applications.

### **14. Java 13 (September 2019)**

* **Key Features**:
  + **Text Blocks (preview)**: Multiline strings with better readability.
  + Switch expressions become a **second preview** feature.
  + Improvements to **ZGC (Z Garbage Collector)**: Supports returning unused memory to the OS.

### **15. Java 14 (March 2020)**

* **Key Features**:
  + **Pattern Matching for instanceof (preview)**: Simplifies casting after instanceof checks.
  + **Records (preview)**: Immutable data classes with minimal boilerplate.
  + **Text Blocks**: Standardized after being in preview.
  + **Helpful NullPointerExceptions**: JVM provides more informative error messages.
  + **Foreign-Memory Access API (preview)**: Allows programs to access memory outside the JVM.

### **16. Java 15 (September 2020)**

* **Key Features**:
  + **Sealed Classes (preview)**: Restricts which classes can extend or implement a class.
  + **ZGC improvements**: Makes Z Garbage Collector ready for production.
  + **Hidden Classes**: Classes that can only be used by frameworks and cannot be directly used by bytecode.

### **17. Java 16 (March 2021)**

* **Key Features**:
  + **Pattern Matching for instanceof**: Now standardized.
  + **Records**: Finalizes the record feature.
  + **Sealed Classes (second preview)**.
  + **Vector API (Incubator)**: Operations for vector computations.
  + Improved **foreign linker API** for calling native code.

### **18. Java 17 (September 2021)** – **LTS** release

* **Key Features**:
  + **Sealed Classes**: Now a standard feature.
  + **Pattern Matching for switch (preview)**.
  + **Deprecation of Applet API**.
  + Removal of the **RMI Activation** mechanism.
  + Improvements in **G1 Garbage Collector** and performance optimizations.

### **19. Java 18 (March 2022)**

* **Key Features**:
  + **Simple Web Server**: Minimal HTTP file serving for easy prototyping.
  + **UTF-8 by Default**: UTF-8 becomes the default charset.
  + **Pattern Matching for switch (third preview)**.
  + Second incubator for **Vector API**.
  + **Foreign Function & Memory API (preview)**: Interact with native libraries and memory outside the JVM.

### **20. Java 19 (September 2022)**

* **Key Features**:
  + **Virtual Threads (preview)**: Improves thread scalability.
  + **Structured Concurrency (incubator)**: Simplifies concurrent code development.
  + Fourth incubator for **Vector API**.
  + **Foreign Function & Memory API (second preview)**.
  + **Pattern Matching for switch (fourth preview)**.

### **21. Java 20 (March 2023)**

* **Key Features**:
  + Virtual Threads further refined.
  + Continued improvements to **Foreign Function & Memory API** and **Pattern Matching**.
  + Advancements in **Project Loom** for scaling concurrency in Java.

### **Java 21 (September 2023)** – **LTS** release

Java 21 is a **Long-Term Support (LTS)** version, meaning it will have extended support for many years. It introduced several important features, building upon previous improvements and continuing to expand Java's capabilities for performance, security, and modern programming paradigms.

* **Virtual Threads (Production Ready)**:
  + Introduced in Java 19 as a preview, Virtual Threads are now fully supported. They allow lightweight concurrency by decoupling tasks from operating system threads, making it easier to write scalable and concurrent applications.
* **Pattern Matching for Switch (Standard)**:
  + After multiple preview iterations, pattern matching for switch statements has become a standard feature. It allows the switch construct to work more effectively with different data types and simplifies code by eliminating the need for explicit type casts.
* **Sequenced Collections**:
  + A new family of interfaces (SequencedCollection, SequencedSet, and SequencedMap) that support methods to access elements in order. It provides a unified view for collections that maintain a predictable iteration order, like lists and linked maps.
* **String Templates (Preview)**:
  + String templates enable the embedding of expressions within string literals in a safe, concise, and flexible manner. It makes string formatting easier and less error-prone than manually concatenating strings and expressions.
* **Unnamed Patterns and Variables**:
  + Java 21 allows the use of unnamed patterns and variables, useful for destructuring objects without binding them to a specific name, reducing boilerplate code.
* **Record Patterns (Final)**:
  + Record patterns allow pattern matching on records, which were introduced in Java 16. This feature improves code readability and conciseness by allowing data classes (records) to be destructured directly within switch and if statements.
* **Scoped Values**:
  + Scoped values provide an alternative to thread-local variables. They are immutable values that can be shared with threads in a more controlled and safer manner, avoiding some of the pitfalls of thread-local storage in concurrent applications.
* **Foreign Function & Memory API (Third Preview)**:
  + Continues to refine the API for accessing native memory and calling native functions. This is particularly useful for Java applications that need to interact with non-Java libraries.
* **Structured Concurrency (Second Incubator)**:
  + Structured Concurrency simplifies the handling of concurrent operations by managing them in a structured way, improving error handling, cancellation, and data aggregation from parallel tasks.
* **Generational ZGC (Experimental)**:
  + The Generational Z Garbage Collector builds on ZGC to improve performance by adding generational collection, which allows the GC to handle short-lived objects more efficiently.

### **Java 22 (September 2023)** – **LTS** release

### 1. **Immutability**

* **What it means**: Scoped values are immutable, meaning that once a value is set, it cannot be changed.
* **Why it's important**: Immutability prevents unintended side effects or race conditions in concurrent applications. It enhances **thread safety** by ensuring that no thread can accidentally or maliciously alter the value, avoiding complex synchronization issues like locking mechanisms.

#### Example:

In traditional **ThreadLocal** variables, you would have to manage synchronization if multiple threads attempt to access or modify the value. With **Scoped Values**, this complexity is reduced because the value is constant once initialized.

### 2. **Automatic Lifecycle Management**

* **What it means**: The scoped value is automatically removed when it goes out of scope (i.e., when the thread or the block of code that uses the value finishes execution).
* **Why it's important**: This simplifies memory management by avoiding common pitfalls like memory leaks, which can occur when data tied to a thread is not properly cleaned up. It provides a built-in mechanism to ensure that scoped values are discarded when no longer needed.

#### Example:

In a scenario where you create a temporary value in a thread, such as session data, the scoped value will automatically be cleaned up when the session (or thread) ends. This makes it easier for developers to manage resources.

### 3. **Thread-Safe Sharing**

* **What it means**: Scoped values can be safely shared between a parent thread and its child threads without fear of the child threads modifying the original value.
* **Why it's important**: This facilitates better coordination between threads, allowing for safe access to shared data. Unlike **ThreadLocal** variables, where each thread maintains its own copy of the value, scoped values can propagate safely across child threads while maintaining immutability.

#### Example:

Imagine a **web server** where each thread represents a request. The main thread could pass a scoped value (like a **user session ID**) to the worker threads processing different parts of the request. Each worker thread would have access to the same session ID without being able to modify it, ensuring consistency.

### Use Case Example: **Scoped Values in Action**

Think of a **shopping cart** metaphor:

* **Customer (Thread)**: Each customer (thread) gets their own shopping cart (scoped value).
* **Shopping Cart**: The cart (value) holds the items that the customer picks. This cart is specific to that customer and can’t be shared or modified by other customers.
* **Immutability**: Once the customer picks an item (sets the value), the items cannot be modified.
* **Automatic Cleanup**: When the customer leaves the store (the thread ends), the cart is automatically discarded, freeing up memory and avoiding clutter.
* **Thread-Safe Sharing**: The customer can ask a store employee (child thread) to help with shopping, and the employee has access to the same shopping cart (scoped value) without modifying the items.

### Key Differences Between **Scoped Values** and **ThreadLocal Variables**:

* **Immutability**: ThreadLocal variables are mutable by default, which can lead to concurrency issues. Scoped values eliminate this risk by being immutable.
* **Propagation**: Scoped values can be explicitly shared with child threads, whereas **ThreadLocal** variables require each thread to maintain its own copy of the data.
* **Lifecycle Management**: Scoped values are automatically cleaned up, reducing the risk of memory leaks, while **ThreadLocal** variables require manual cleanup to prevent such issues.

### Conclusion

**Scoped Values** are a powerful new tool introduced in **JDK 22** that significantly improve thread safety, immutability, and lifecycle management when dealing with shared data across threads. They simplify concurrent programming in Java by providing a cleaner and more efficient alternative to **ThreadLocal** variables.

This feature is especially valuable for applications that rely heavily on concurrency and threading, such as **web servers, distributed systems, and parallel processing frameworks**.

**2. Features of Java**

1. **Simple**: Java’s syntax is clear and easy to learn for developers who have experience in other languages like C and C++. Java removes many complex features, such as pointers and operator overloading, making it easier to use.
2. **Object-Oriented**: Everything in Java revolves around classes and objects. It follows principles such as inheritance, polymorphism, abstraction, and encapsulation.
3. **Platform-Independent**: Java programs are compiled into bytecode that runs on the Java Virtual Machine (JVM), making them independent of operating systems. "Write Once, Run Anywhere (WORA)" is a key principle.
4. **Secure**: Java has built-in security features, such as bytecode verification, access control, and exception handling, which provide a robust environment for building secure applications.
5. **Robust**: Java has strong memory management through automatic garbage collection, exception handling, and type checking, reducing the likelihood of crashing or memory leaks.
6. **Multithreaded**: Java supports concurrent programming through multithreading, allowing multiple threads to run simultaneously. This is useful in applications like gaming, multimedia, and web servers.
7. **High Performance**: Though not as fast as C or C++, Java’s performance is boosted by the use of Just-In-Time (JIT) compilation and efficient garbage collection.
8. **Distributed**: Java supports distributed computing, allowing users to work on multiple systems connected to a network. Java’s networking capabilities are used for building enterprise-scale applications.
9. **Dynamic**: Java is highly dynamic and adaptable, supporting dynamic loading of classes, functions, and libraries during runtime.
10. **Portable**: Because Java compiles into bytecode, it can run on any system with a JVM, making it portable across different hardware architectures.

**3. Career Scope of Java**

Java offers vast career opportunities. Some career paths include:

* **Backend Developer**: Specializing in server-side logic, database interactions, and APIs using Java frameworks like Spring Boot.
* **Full Stack Developer**: Working with both frontend (using technologies like Angular or React) and backend (Java-based frameworks).
* **Android Developer**: Building mobile applications using Java or Kotlin (which runs on the JVM).
* **Example Projects**:
  + **Banking Applications**: Java is used for transaction processing, financial services, and ATM software.
  + **Web Portals**: Java is often used to build websites or backend APIs for websites like LinkedIn or e-commerce platforms.

**4. Types of Applications Built with Java**

* **Core Java (JSE)**: Used for building standalone applications like desktop software (e.g., text editors, media players).
  + **Example**: A simple note-taking application using Swing (Java's GUI framework).
* **Advanced Java (JEE)**: Suitable for web applications, enterprise systems, distributed systems, and more.
  + **Example**: A web-based e-commerce system using JSP (Java Server Pages), Servlets, and databases.
* **Spring Boot**: A modern framework for building scalable microservices and web applications.
  + **Example**: A REST API for managing employee data (CRUD operations) with endpoints like /employees.

**5. What is a Compiler, Interpreter, and Assembler?**

* **Compiler**: Converts entire high-level code (like Java) into machine code or bytecode before execution.
  + **Example**: Java uses the javac compiler to convert .java files into .class bytecode files.
* **Interpreter**: Executes code line-by-line. Java uses the JVM to interpret the bytecode and execute it on the system.
  + **Example**: The JVM interprets bytecode and runs it on any OS.
* **Assembler**: Converts assembly language (a low-level language) into machine code that the processor understands.
  + **Example**: In system programming, an assembler might be used to write hardware-specific code.

**6. Low-Level, High-Level, and Middle-Level Languages**

* **Low-Level Language**: These languages (e.g., Assembly) are close to machine language and are hardware-specific.
  + **Example**: Assembly language instructions for arithmetic operations.
* **High-Level Language**: Easier to read and write by humans (e.g., Java, Python). These are abstracted from hardware.
  + **Example**: In Java, System.out.println("Hello"); prints to the console without needing to manage CPU or memory directly.
* **Middle-Level Language**: These languages (e.g., C) provide features of both low-level and high-level languages, allowing for more control while still being easier to write than assembly.
  + **Example**: Writing a program in C that manipulates memory addresses.

**7. Functional, Procedural, and Object-Oriented Programming**

* **Functional Programming**: A style of programming that treats computation as the evaluation of mathematical functions without side effects.
  + **Example in Java** (with lambdas):

List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

numbers.stream().map(n -> n \* 2).forEach(System.out::println);

* **Procedural Programming**: Focuses on procedures or routines. Java supports this through functions and loops.
  + **Example**:

public class Sum {

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

return a + b;

}

}

* **Object-Oriented Programming (OOP)**: Java is based on OOP principles like inheritance, encapsulation, abstraction, and polymorphism.
  + **Example**:

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}

}

**8. What is Client and Server?**

* **Client**: The device (or software) that requests resources or services from a server (e.g., a web browser requesting a webpage).
  + **Example**: A web browser sends a request to a web server for a webpage.
* **Server**: A machine or software that provides services or resources to the client.
  + **Example**: A web server hosts and serves HTML pages, APIs, or media files to the client.

**9. What is Localhost?**

**Localhost** is the default hostname that refers to the local computer. It is often used for testing and developing applications on a local machine.

* **Example**: If a developer runs a web server on their local machine, they can access it by visiting http://localhost:8080 in a browser.

**10. Tools Used in Java Development and Why**

In real-life software development, tools play a crucial role in streamlining processes, improving efficiency, and ensuring high-quality outcomes. Here are a few reasons why tools are important, along with real-life examples:

* **Integrated Development Environments (IDEs)**: Tools like Eclipse and IntelliJ IDEA provide features like syntax highlighting, auto-completion, and debugging.
  + **Example**: Debugging Java code in IntelliJ.
* **Build Tools**: Maven and Gradle manage project dependencies and automate build processes.
  + **Example**: Using Maven to include Spring Boot dependencies in a project.
* **Version Control Systems**: Git and GitHub are used for tracking code changes, collaboration, and managing code versions.
  + **Example**: Committing and pushing Java code changes to GitHub.

#### **1. Postman**

**Postman** is a widely used API development tool that allows developers to design, test, and document APIs without needing to write any additional code. It simplifies testing RESTful web services by providing a GUI for sending HTTP requests and viewing the responses.

* **Why Use Postman?**
  + To test Java backend APIs (built with Spring Boot, for example) by making GET, POST, PUT, DELETE requests.
  + To debug APIs before integrating them with frontend applications.
* **Example Usage:**
  + You can send a POST request to http://localhost:8080/api/employees with a JSON payload to add a new employee to the database.

#### **2. Jira**

**Jira** is an issue-tracking tool developed by Atlassian that is widely used for agile project management, bug tracking, and task management. It helps teams manage development tasks, track bugs, and coordinate work.

* **Why Use Jira?**
  + To track and manage project tasks, bugs, and development progress.
  + To manage sprints in Agile and track stories assigned to developers.
* **Example Usage:**
  + A developer uses Jira to update the status of a bug they are fixing in a Spring Boot application. A task might be tagged as "In Progress" and then "Completed" once done.

#### **3. PuTTY**

**PuTTY** is a free and open-source SSH and telnet client. It allows developers to securely connect to remote servers and perform command-line tasks.

* **Why Use PuTTY?**
  + To connect to remote Linux servers where Java applications are hosted.
  + To deploy or troubleshoot server-side Java applications.
* **Example Usage:**
  + Use PuTTY to SSH into a remote server and check the status of a Java Spring Boot microservice using commands like systemctl status myservice.

#### **4. Log4j / SLF4J**

**Log4j** and **SLF4J** are logging libraries used in Java to log messages during application runtime. They help developers monitor application behavior and troubleshoot issues.

* **Why Use Log4j / SLF4J?**
  + To generate logs for debugging or tracking application flow.
  + To log error messages, warnings, or other useful information in a structured way.
* **Example Usage:**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class EmployeeService {

private static final Logger logger = LoggerFactory.getLogger(EmployeeService.class);

public void addEmployee(Employee emp) {

logger.info("Adding employee: {}", emp.getName());

}

}

#### **5. JUnit / Mockito / PowerMockito**

* **JUnit** is a popular framework for unit testing Java applications.
* **Mockito** is a mocking framework used in conjunction with JUnit to mock dependencies in unit tests.
* **PowerMockito** extends Mockito and allows testing code with static methods, constructors, etc., which are otherwise hard to mock.
* **Why Use JUnit / Mockito / PowerMockito?**
  + To ensure your code works as expected by writing unit tests.
  + Mockito is useful to simulate the behavior of classes your code interacts with, making it easier to test code in isolation.
  + PowerMockito is used for mocking static methods or final classes, which are otherwise difficult to test.
* **Example Usage:**

@RunWith(MockitoJUnitRunner.class)

public class EmployeeServiceTest {

@Mock

private EmployeeRepository employeeRepository;

@InjectMocks

private EmployeeService employeeService;

@Test

public void testAddEmployee() {

Employee emp = new Employee("John");

employeeService.addEmployee(emp);

verify(employeeRepository, times(1)).save(emp);

}

}

#### 6. **Swagger**

**Swagger** is an open-source tool used to document and test RESTful APIs. It automatically generates interactive API documentation from your Java code.

* **Why Use Swagger?**
  + To provide clear, interactive API documentation.
  + It enables testing APIs directly from the documentation.
* **Example Usage:**
  + Spring Boot applications can integrate Swagger with the springfox-swagger2 and springfox-swagger-ui libraries. After integration, Swagger provides an interactive UI where users can test API endpoints.

swagger: '2.0'

info:

description: API for managing employees

paths:

/employees:

get:

summary: Retrieve a list of employees

#### **7. FileZilla**

**FileZilla** is an FTP client that allows developers to transfer files between their local machine and a remote server.

* **Why Use FileZilla?**
  + To upload Java application files or logs to a remote server.
  + To download files or logs from a remote server for debugging.
* **Example Usage:**
  + A developer can use FileZilla to transfer a WAR file from their local machine to the server where the Java application is hosted.

#### **8. WinSCP**

**WinSCP** is another tool for file transfer, primarily used on Windows. It supports FTP, SFTP, and SCP protocols.

* **Why Use WinSCP?**
  + To securely transfer files between a local machine and a remote server (similar to FileZilla).
  + WinSCP also integrates with PuTTY for easier remote command execution after transferring files.
* **Example Usage:**
  + After deploying a Java application on a Linux server, you can use WinSCP to access log files for analysis or transfer necessary configuration files.

#### **9. Linux**

**Linux** is a widely used operating system in server environments, and many Java applications are deployed on Linux-based servers.

* **Why Use Linux?**
  + Linux is preferred for server-side applications because it is stable, secure, and customizable.
  + Java applications are often deployed on Linux servers in production environments.
* **Example Usage:**
  + A developer might use Linux commands to start or stop a Java Spring Boot microservice running on a server:

sudo systemctl start my-java-service

### 11. ****What is Full Stack, Frontend, and Backend?****

A full-stack developer handles both **frontend** and **backend** development.

* **Frontend**: Refers to the part of the application that users interact with directly (UI/UX).
  + Technologies: HTML, CSS, JavaScript, Angular, React
  + **Example**: A webpage designed using HTML, CSS, and JavaScript.
* **Backend**: Refers to the server-side of an application. It handles data processing, database interactions, and business logic.
  + Technologies: Java (Spring Boot), Node.js, Python, Databases
  + **Example**: A REST API created in Java using Spring Boot that interacts with a MySQL database.
* **Full Stack**: A developer skilled in both frontend and backend technologies can build complete applications.

### 12. ****Machine Code and Human Code Conversion****

**Machine code** consists of binary instructions (1s and 0s) that are executed directly by a computer's CPU. High-level programming languages like Java are easier for humans to read and write but must be compiled into machine code (or bytecode in the case of Java) to run on the machine.

* **Example of Conversion**:
  + The Java code:

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

is compiled into bytecode:

0xB2 0x00 0x04 0xB6 0x00 0x05 0xB1

* **Why Convert?**
  + Computers only understand machine code (binary instructions), so any code written by a human in high-level languages must be compiled into machine code.

These tools and concepts play a critical role in the development, testing, and deployment of Java applications. By understanding and utilizing these tools effectively, developers can streamline the process and ensure the quality of their software.

### **How to Use Java?**

To develop applications using Java, follow these steps:

#### **Setting Up Java Environment**

1. **Download and Install JDK (Java Development Kit)**:
   * The JDK contains the Java compiler (javac), the Java runtime environment (JRE), and libraries necessary for development.
   * Download from [Oracle](https://www.oracle.com/java/technologies/javase-downloads.html) or use OpenJDK.
2. **Set Up Environment Variables**:
   * Add Java’s bin folder to your system’s PATH environment variable so that Java commands like javac and java can be used from the command line.
3. **Write Your First Program**:
   * Java programs are written in .java files and compiled into .class files.

**Example Program:**

public class HelloWorld {

public static void main(String[] args) {

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

}

}

1. **Compile and Run Java Program**:
   * **Compile**: javac HelloWorld.java
   * **Run**: java HelloWorld

#### **Key Components in Java Development**

* **JVM (Java Virtual Machine)**: Runs Java bytecode. Different platforms have their own JVM implementation.
* **JRE (Java Runtime Environment)**: Contains the JVM and libraries required to run Java applications.
* **JDK (Java Development Kit)**: Includes the JRE and additional tools such as the compiler (javac) for development.

### **When to Use Java?**

Java is ideal for a variety of application development scenarios. Here are common use cases:

* **Enterprise Applications**: Java’s scalability and robustness make it a preferred choice for large-scale, enterprise-level applications.
  + **Examples**: Banking systems, ERP systems, CRM systems.
* **Web Applications**: Frameworks like **Spring** and **Hibernate** simplify Java-based web development.
  + **Examples**: E-commerce websites, financial portals.
* **Android Development**: Java is the primary language used for developing Android apps. Google’s Android SDK is built using Java.
  + **Examples**: Mobile apps, games, tools.
* **Distributed Systems**: Java supports the development of distributed applications using technologies like **RMI (Remote Method Invocation)** and **EJB (Enterprise JavaBeans)**.
  + **Examples**: Cloud services, microservices architectures.
* **Big Data Technologies**: Java is used in tools like **Hadoop**, **Apache Kafka**, and **Apache Spark** for managing big data.
  + **Examples**: Data analysis tools, machine learning platforms.
* **Scientific and Financial Applications**: With its high performance and security, Java is used in scientific computing and high-frequency trading platforms.

### **5. Where to Use Java?**

Java can be used in a variety of platforms and industries:

#### **1. Web Development:**

* **Frameworks**: Spring, Hibernate, Struts.
* **Application Servers**: Apache Tomcat, JBoss, GlassFish.
* **Use Case**: Building REST APIs, microservices, and enterprise-level web applications.

#### **2. Android Development:**

* **Tools**: Android SDK, Android Studio.
* **Use Case**: Developing native Android apps for mobile devices.

#### **3. Enterprise Applications:**

* **Tools**: Java EE, Spring Boot, EJB.
* **Use Case**: Large-scale enterprise applications such as CRM, ERP systems.

#### **4. Cloud Computing:**

* **Tools**: AWS SDK for Java, Google Cloud, Microsoft Azure SDKs.
* **Use Case**: Building scalable cloud-based applications using microservices and distributed systems.

#### **5. Big Data Technologies:**

* **Frameworks**: Hadoop, Apache Spark, Apache Kafka.
* **Use Case**: Processing and analyzing big data in real time.

#### **6. IoT and Embedded Systems:**

* **Tools**: Java ME (Micro Edition), OpenJDK.
* **Use Case**: Developing applications for small devices such as sensors and IoT gateways.

#### **7. Financial Services:**

* **Tools**: Java SE, multithreading libraries.
* **Use Case**: Building secure, real-time financial applications such as trading platforms, banking software.

### 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.

### Notes:

* **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.

**What is an Identifier?**

An identifier is a name given to elements in a program, such as variables, methods, classes, objects, etc. In Java, identifiers are used to uniquely name elements for easy reference in code.

**Rules for Naming Identifiers:**

1. **Allowed Characters**: Identifiers can only consist of letters (a-z, A-Z), digits (0-9), underscore (\_), and dollar sign ($).
2. **No Reserved Keywords**: Keywords (like int, class, static, etc.) cannot be used as identifiers.
3. **Case-sensitive**: Java treats identifiers as case-sensitive, meaning MyClass and myclass are considered different.
4. **Cannot Begin with a Digit**: An identifier cannot start with a number, but it can contain numbers after the first letter.

**Example**:

* + Valid: myVariable, counter1, \_totalSum, $mainClass.
  + Invalid: 1variable (cannot start with a digit), class (keyword), total#sum (special character not allowed).

**Types of Identifiers in Java**

Java identifies various elements in a program, such as:

1. **Variable Names**: For storing data.
   * Example: int age;
2. **Method Names**: For defining actions.
   * Example: void calculateSum()
3. **Class Names**: For defining the blueprint of an object.
   * Example: class Student {}
4. **Object Names**: Instance of a class.
   * Example: Student student1 = new Student();
5. **Constants**: Final variables are usually written in uppercase.
   * Example: final int MAX\_COUNT = 100;

**Best Practices for Naming Identifiers:**

1. **Meaningful Names**: Choose meaningful names that reflect the purpose of the variable, method, or class. For example, instead of naming a variable x, use age, price, or name to enhance code readability.

**Example**:

// Good identifier name

int numberOfStudents;

// Bad identifier name

int n;

1. **Follow Camel Case**: It's a common convention to use **camel case** for variable and method names in Java. The first letter is lowercase, and subsequent words start with an uppercase letter.
   * **Class Names**: Start with an uppercase letter.
     + Example: StudentData
   * **Method and Variable Names**: Start with a lowercase letter.
     + Example: calculateArea
2. **Avoid Lengthy Names**: While meaningful names are important, extremely long identifiers should be avoided for simplicity.

**Practical Example 1: Variable Identifiers**

public class IdentifierExample {

public static void main(String[] args) {

int age = 25; // 'age' is the identifier

String name = "John"; // 'name' is the identifier

double salary = 50000.50; // 'salary' is the identifier

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

}

}

**Output**:

Name: John, Age: 25, Salary: 50000.5

In this example, age, name, and salary are the identifiers for different variables.

**Practical Example 2: Class and Method Identifiers**

class Calculator { // 'Calculator' is the identifier for the class

public int add(int num1, int num2) { // 'add' is the method identifier

return num1 + num2;

}

}

public class Main {

public static void main(String[] args) {

Calculator calc = new Calculator(); // 'calc' is the identifier for the object

int sum = calc.add(10, 20); // 'sum' is the identifier

System.out.println("Sum: " + sum);

}

}

**Output**:

Sum: 30

Here, we have:

* Calculator as the class identifier.
* add as the method identifier.
* num1, num2 as parameter identifiers.
* calc as an object identifier.
* sum as a variable identifier.

**Common Errors with Identifiers:**

1. **Using a reserved keyword**: Trying to use Java keywords like class, int, new as identifiers.

int new = 5; // Error! 'new' is a keyword.

1. **Starting with a digit**: An identifier cannot start with a digit.

int 2value = 100; // Error! Cannot start with a digit.

1. **Using special characters**: Identifiers should only contain letters, digits, underscore, or dollar sign.

int value# = 100; // Error! Special character '#' not allowed.

**1. Allowed Characters:**

Identifiers can only consist of letters (a-z, A-Z), digits (0-9), underscore (\_), and dollar sign ($). Any other special characters are not allowed.

public class IdentifierExample {

public static void main(String[] args) {

int validVariable = 10; // valid identifier (letters)

int \_validVariable = 20; // valid identifier (underscore)

int $validVariable = 30; // valid identifier (dollar sign)

int valid123 = 40; // valid identifier (contains digits after letters)

// Example usage

System.out.println(validVariable); // Output: 10

System.out.println(\_validVariable); // Output: 20

System.out.println($validVariable); // Output: 30

System.out.println(valid123); // Output: 40

// Invalid identifiers:

// int invalid-var = 50; // Error: '-' not allowed

// int valid!@ = 60; // Error: '!' and '@' are not allowed

}

}

**2. No Reserved Keywords:**

Java keywords such as int, class, static, void, etc., cannot be used as identifiers.

public class IdentifierKeywordExample {

public static void main(String[] args) {

// int class = 100; // Error: 'class' is a keyword

// int static = 200; // Error: 'static' is a keyword

// Correct usage of valid identifiers:

int myClass = 100; // Valid (not using reserved keyword)

int myStatic = 200; // Valid (not using reserved keyword)

System.out.println(myClass); // Output: 100

System.out.println(myStatic); // Output: 200

}

}

**3. Case-Sensitive:**

Java is case-sensitive, meaning MyClass and myclass are treated as two different identifiers.

public class CaseSensitiveExample {

public static void main(String[] args) {

int MyClass = 10; // Valid identifier

int myclass = 20; // Different identifier due to case sensitivity

System.out.println(MyClass); // Output: 10

System.out.println(myclass); // Output: 20

}

}

In this example, MyClass and myclass are considered two different variables due to their case difference.

**4. Cannot Begin with a Digit:**

An identifier cannot start with a digit, but it can contain digits after the first letter.

public class DigitIdentifierExample {

public static void main(String[] args) {

int valid123 = 100; // Valid identifier (starts with a letter, contains digits)

// int 123valid = 200; // Error: Cannot start with a digit

System.out.println(valid123); // Output: 100

}

}

**Combining the Concepts:**

In this example, let's create a small program that uses valid identifiers, demonstrates case-sensitivity, and adheres to the rules we've discussed.

public class IdentifierDemo {

public static void main(String[] args) {

// Correct identifier examples:

int age = 25; // Valid

int $salary = 50000; // Valid (dollar sign allowed)

int \_year2024 = 2024; // Valid (underscore and digits allowed)

// Case-sensitivity demonstration:

int Age = 30; // Different from 'age'

int AGE = 35; // Different from 'age' and 'Age'

System.out.println(age); // Output: 25

System.out.println(Age); // Output: 30

System.out.println(AGE); // Output: 35

// Invalid examples:

// int 123name = 50; // Error: Cannot start with a digit

// int public = 60; // Error: 'public' is a reserved keyword

}

}

**Explanation:**

* **Allowed Characters**: We use a combination of letters, digits, underscore, and dollar sign in identifiers like \_year2024, $salary.
* **No Reserved Keywords**: We ensure that no Java reserved keywords are used, such as public, class, etc.
* **Case-sensitive**: Variables age, Age, and AGE are treated as three different identifiers due to case sensitivity.
* **Cannot Begin with a Digit**: The program avoids starting any identifier with a digit (123name would cause an error).

These examples give a comprehensive understanding of how identifiers work in Java, following all the rules and best practices.

### ****Variables in Java****

Variables in Java act as containers to store data values. They must be declared with a **data type** that determines the kind of data the variable can hold. Java variables can be classified into three main categories based on where they are declared and how they behave.

#### **Types of Variables**

1. **Local Variables:**
   * **Definition:** Declared within a method or a block and are only accessible within that method/block.
   * **Scope:** Limited to the method/block where declared.
   * **Initialization:** Must be initialized before use.
   * **Example:**

public class LocalVariableExample {

public void display() {

int localVar = 100; // Local variable

System.out.println("Local Variable: " + localVar);

}

}

* + **Output:**

Local Variable: 100

1. **Instance Variables:**
   * **Definition:** Declared inside a class but outside any method. They are tied to a specific object (instance of the class).
   * **Scope:** Throughout the class as long as the object exists.
   * **Access:** Accessed using an object of the class.
   * **Example:**

public class InstanceVariableExample {

// Instance variables

int empId;

String empName;

public static void main(String[] args) {

// Creating an object

InstanceVariableExample emp1 = new InstanceVariableExample();

emp1.empId = 101;

emp1.empName = "Alice";

System.out.println("Employee ID: " + emp1.empId);

System.out.println("Employee Name: " + emp1.empName);

}

}

* + **Output:**

Employee ID: 101

Employee Name: Alice

1. **Static Variables:**
   * **Definition:** Declared with the static keyword. Shared by all instances of the class, meaning there is only one copy of the static variable in memory.
   * **Scope:** Associated with the class itself, not with objects.
   * **Access:** Can be accessed directly using the class name.
   * **Example:**

public class StaticVariableExample {

// Static variable

static String company = "Tech Solutions";

public static void main(String[] args) {

System.out.println("Company: " + company);

}

}

* + **Output:**

Company: Tech Solutions

### ****Variable Declaration and Initialization****

In Java, variables can be declared and initialized at the same time or separately.

* **Declaration:** Assigns a data type and a variable name.
* **Initialization:** Assigns an initial value to the variable.

**Example:**

public class VariableExample {

// Instance variable

String name;

// Static variable

static String company = "Tech Corp";

public static void main(String[] args) {

// Local variable

int age = 30;

System.out.println("Company: " + company);

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

// Creating an object to access the instance variable

VariableExample emp = new VariableExample();

emp.name = "John";

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

}

}

**Output:**

Company: Tech Corp

Age: 30

Employee Name: John

### ****Scope of Variables****

1. **Local Scope:**
   * Variables declared inside a method or block are accessible only within that method/block.
2. **Instance Scope:**
   * Instance variables exist as long as the object that contains them exists.
3. **Static Scope:**
   * Static variables are available as long as the class is loaded into memory, shared across all instances.

**Example:**

public class VariableScopeExample {

// Static variable

static String company = "Tech Global";

// Instance variable

int empId;

public void show() {

// Local variable

String empName = "Alice";

System.out.println("Company: " + company); // Accessing static variable

System.out.println("Employee ID: " + empId); // Accessing instance variable

System.out.println("Employee Name: " + empName); // Accessing local variable

}

public static void main(String[] args) {

// Creating an object

VariableScopeExample emp1 = new VariableScopeExample();

emp1.empId = 101;

emp1.show();

}

}

**Output:**

Company: Tech Global

Employee ID: 101

Employee Name: Alice

Example:

**package** com.weekend.class1.core.java1;

**public** **class** VariablePractical {

String str="Amarjeet";// instance variable , how to call instance variable

**static** String *name*="Usha"; // static variable , how to call static variable guys

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

**int** x=10; //local variable , how to call local variable

//this is for the local variable

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

//below is for instance one

VariablePractical anupma=**new** VariablePractical();

System.***out***.println(anupma.str);

//for static one

VariablePractical objectName= **new** VariablePractical();

System.***out***.println(objectName.*name*);

System.***out***.println("this is static caling: "+*name*);

System.***out***.println("always used with calss or interface neme: "+VariablePractical.*name*);

//keep in the mind there is three type of variable local, static, instance

}

}

### ****Data Types in Java****

Java is a strongly typed language, meaning every variable must have a data type, which specifies the type and size of data it can store. Java provides two categories of data types:

Or Which type of data you are going to assign to particular variable called data type

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

These are the most basic data types and are predefined in Java.

* **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 fractional numbers.
* **double:** 8 bytes, stores double precision fractional numbers.
* **char:** 2 bytes, stores a single character (Unicode).
* **boolean:** 1 bit, stores true or false.

**Example:**

public class DataTypeExample {

public static void main(String[] args) {

// Primitive data types

int age = 25; // Integer type

double salary = 45000.75; // Floating-point type

char grade = 'A'; // Character type

boolean isEmployed = true; // Boolean type

// Printing values

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

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

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

System.out.println("Employed: " + isEmployed);

}

}

**Output:**

Age: 25

Salary: 45000.75

Grade: A

Employed: true

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

These refer to objects and are created using defined classes.

* **Examples:** Classes, arrays, interfaces, and strings.
* Unlike primitive data types, reference types do not hold the data directly but reference the memory location where the object is stored.

### ****Conclusion****

* **Variables** serve as containers for data, and their scope determines their lifetime and accessibility.
* **Data types** define the kind of values a variable can hold.
* Understanding the scope of variables and their types is fundamental to developing efficient Java programs.

### 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.

### Notes:

* **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.

**Notes:**

| **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:**

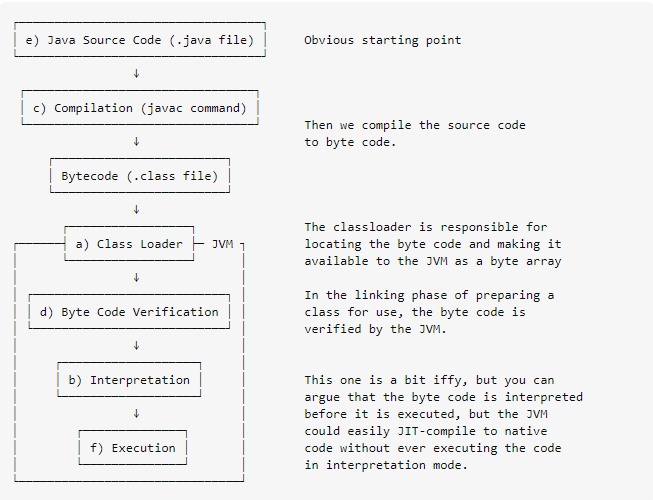
* **Shutdown Hooks:** The JVM provides a mechanism for executing code before the JVM terminates (e.g., releasing resources, saving state).
* **Resource Management:** Ensures all resources (e.g., file handles, network connections) are properly closed.

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

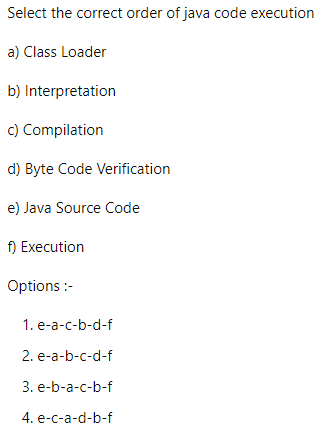
1. **Class Loading:**
   * Class files are loaded into memory by the class loader.
   * The class loader handles the loading, linking, and initialization of classes.
2. **Memory Management:**
   * The JVM allocates memory for classes, objects, and thread-specific data.
   * The garbage collector manages 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/>

**Unit-2 Notes**

## **Java Arrays**

### **1. Introduction**

* An array is a data structure that stores a fixed-size sequential collection of elements of the same type.
* Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
* We can access the array through indexing.

Advantage of Array:

* Code Optimization
* Random Access
* Easy to Use and Ease of Iteration

Disadvantage of Array:

* Fixed in size
* Only can store homogenious data items(same time of data)

### **2. One-Dimensional Array**

#### **2.1. Declaration and Initialization**

* **Declaration**: Specifies the type of elements and the name of the array.

int[] arr; // Declares an array of integers

int []arr;

int arr[];

* **Initialization**: Allocates memory for the array and optionally initializes its elements.

arr = new int[5]; // Creates an array of size 5

* **Declaration and Initialization in a Single Line**:

int[] arr = new int[5]; // Creates an array of size 5 with default values (0 for integers)

* **Array Initialization with Values**:

int[] arr = {1, 2, 3, 4, 5}; // Creates and initializes the array

#### **2.2. Accessing Array Elements**

* Access elements using the index (zero-based).

int firstElement = arr[0]; // Accesses the first element of the array

#### **2.3. Iterating Over an Array**

* **Using a for loop**:

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

System.out.println(arr[i]);

}

* **Using an enhanced for loop**:

for (int value : arr) {

System.out.println(value);

}

#### **2.4. Common Methods**

* **length**: Returns the number of elements in the array.

int size = arr.length;

### **3. Multidimensional Arrays**

#### **3.1. Declaration and Initialization**

* **Declaration**:

int[][] matrix; // Declares a two-dimensional array

* **Initialization**:

matrix = new int[3][4]; // Creates a 3x4 matrix

* **Declaration and Initialization in a Single Line**:

int[][] matrix = new int[3][4]; // Creates a 3x4 matrix

* **Array Initialization with Values**:

int[][] matrix = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12}

};

#### **3.2. Accessing Elements**

* Access elements using two indices (row and column).

int value = matrix[1][2]; // Accesses the element in the 2nd row and 3rd column

#### **3.3. Iterating Over a Multidimensional Array**

* **Using nested for loops**:

for (int i = 0; i < matrix.length; i++) { // Iterate over rows

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

System.out.println(matrix[i][j]);

}

}

### **Java Strings:**

#### **1. What is a String?**

In Java, a String is an object that represents a sequence of characters. It is used to store and manipulate text. Java String objects are immutable, meaning once created, their values cannot be changed.

#### **2. Ways to Create String Objects**

1. **Using String Literals**:
   * String str1 = "Hello";
   * Stored in the String Constant Pool. Reuses existing instances with the same value.
2. **Using the new Keyword**:
   * String str2 = new String("Hello");
   * Creates a new String object in the heap memory.
3. **Using Character Arrays**:
   * char[] chars = {'H', 'e', 'l', 'l', 'o'};
   * String str3 = new String(chars);
4. **Using Byte Arrays**:
   * byte[] bytes = {65, 66, 67};
   * String str4 = new String(bytes);

#### **3. Advantages and Disadvantages of Strings**

**Advantages**:

1. **Immutability**:
   * Strings are immutable, ensuring thread safety and allowing string interning.
2. **Ease of Use**:
   * Strings come with many built-in methods for manipulation and comparison.
3. **Memory Efficiency**:
   * Strings in the constant pool save memory by reusing immutable string instances.

**Disadvantages**:

1. **Performance Overhead**:
   * Frequent modifications (e.g., concatenation) lead to creation of new String objects, impacting performance.
2. **Memory Consumption**:
   * Excessive string object creation can increase memory usage and garbage collection overhead.

#### **4. Immutability in Java**

**Immutability** means that once a String object is created, its state cannot be modified.

**Example of Immutability**:

public class ImmutableStringExample {

public static void main(String[] args) {

String str1 = "Hello";

String str2 = str1.concat(" World");

// str1 remains unchanged

System.out.println("Original String: " + str1); // Output: "Hello"

System.out.println("Concatenated String: " + str2); // Output: "Hello World"

}

}

#### **5. Heap and String Constant Pool**

**Heap**:

* Dynamic memory allocation area where objects are created using the new keyword.

**String Constant Pool**:

* Special memory area where Java stores string literals to optimize memory usage.

**Example**:

public class HeapAndPoolExample {

public static void main(String[] args) {

String str1 = "Java"; // Stored in String Constant Pool

String str2 = new String("Java"); // Creates a new object in the Heap

System.out.println(str1 == str2); // Output: false (different references)

System.out.println(str1.equals(str2)); // Output: true (same content)

}

}

#### **6. Difference Between** == **and** .equals()

* **== Operator**:
  + Compares memory references, not content.
* **.equals() Method**:
  + Compares content of the strings.

**Example**:

public class EqualsVsDoubleEquals {

public static void main(String[] args) {

String str1 = "Hello";

String str2 = new String("Hello");

System.out.println(str1 == str2); // Output: false (different references)

System.out.println(str1.equals(str2)); // Output: true (same content)

}

}

#### **7. String Methods**

1. **length()**:
   * Returns the length of the string.
   * int len = str.length();
   * **Example**: "Hello".length() returns 5.
2. **isEmpty()**:
   * Checks if the string is empty.
   * boolean empty = str.isEmpty();
   * **Example**: "".isEmpty() returns true.
3. **trim()**:
   * Removes leading and trailing whitespace.
   * String trimmed = str.trim();
   * **Example**: " Hello ".trim() returns "Hello".
4. **charAt(int index)**:
   * Returns the character at the specified index.
   * char ch = str.charAt(0);
   * **Example**: "Hello".charAt(0) returns 'H'.
5. **concat(String str)**:
   * Concatenates the specified string to the end.
   * String concatenated = str.concat(" World");
   * **Example**: "Hello".concat(" World") returns "Hello World".
6. **split(String regex)**:
   * Splits the string around matches of the given regular expression.
   * String[] parts = str.split(",");
   * **Example**: "a,b,c".split(",") returns ["a", "b", "c"].
7. **indexOf(String str)**:
   * Returns the index of the first occurrence of the specified substring.
   * int index = str.indexOf("lo");
   * **Example**: "Hello".indexOf("lo") returns 3.
8. **lastIndexOf(String str)**:
   * Returns the index of the last occurrence of the specified substring.
   * int lastIndex = str.lastIndexOf("o");
   * **Example**: "Hello".lastIndexOf("o") returns 4.
9. **endsWith(String suffix)**:
   * Checks if the string ends with the specified suffix.
   * boolean ends = str.endsWith("llo");
   * **Example**: "Hello".endsWith("llo") returns true.
10. **startsWith(String prefix)**:
    * Checks if the string starts with the specified prefix.
    * boolean starts = str.startsWith("He");
    * **Example**: "Hello".startsWith("He") returns true.
11. **equals(String anotherString)**:
    * Compares the string to the specified object for equality.
    * boolean equals = str.equals("Hello");
    * **Example**: "Hello".equals("Hello") returns true.
12. **equalsIgnoreCase(String anotherString)**:
    * Compares the string to the specified object, ignoring case considerations.
    * boolean equalsIgnoreCase = str.equalsIgnoreCase("HELLO");
    * **Example**: "Hello".equalsIgnoreCase("HELLO") returns true.
13. **contains(CharSequence sequence)**:
    * Checks if the string contains the specified sequence of characters.
    * boolean contains = str.contains("ell");
    * **Example**: "Hello".contains("ell") returns true.
14. **replace(CharSequence target, CharSequence replacement)**:
    * Replaces each substring of this string that matches the specified target sequence with the specified replacement sequence.
    * String replaced = str.replace("l", "x");
    * **Example**: "Hello".replace("l", "x") returns "Hexxo".
15. **replaceFirst(String target, String replacement)**:
    * Replaces the first substring that matches the given regular expression with the given replacement.
    * String replacedFirst = str.replaceFirst("l", "x");
    * **Example**: "Hello".replaceFirst("l", "x") returns "Hexlo".
16. **replaceAll(String target, String replacement)**:
    * Replaces each substring that matches the given regular expression with the given replacement.
    * String replacedAll = str.replaceAll("l", "x");
    * **Example**: "Hello".replaceAll("l", "x") returns "Hexxo".
17. **toLowerCase()**:
    * Converts all characters in the string to lowercase.
    * String lower = str.toLowerCase();
    * **Example**: "Hello".toLowerCase() returns "hello".
18. **toUpperCase()**:
    * Converts all characters in the string to uppercase.
    * String upper = str.toUpperCase();
    * **Example**: "Hello".toUpperCase() returns "HELLO".
19. **join(CharSequence delimiter, CharSequence... elements)**:
    * Joins the given elements into a single string separated by the specified delimiter.
    * String joined = String.join(", ", "a", "b", "c");
    * **Example**: String.join(", ", "a", "b", "c") returns "a, b, c".
20. **subSequence(int start, int end)**:
    * Returns a new character sequence that is a subsequence of this sequence.
    * CharSequence subSeq = str.subSequence(1, 4);
    * **Example**: "Hello".subSequence(1, 4) returns "ell".

#### **Complete Example with All Methods**

**public** **class** StringMethods {

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

String str1 = "AMARJET KUMAR singh";

String str2 = "amarjeet Kumar Singh";

// 1. length() --> Used to find the length of the string.

System.***out***.println(str1.length());

// 2. concat() --> Used to combine two string.

String str = str1.concat("From Bihar");

System.***out***.println("Combining two string: " + str);

// 3. indexOf() --> Used to find index of value.

**int** index1 = str1.indexOf("S");

System.***out***.println("first index: " + index1);

// 4.lastIndexOf()--> Used to find last index of the element.

**int** lastIndexOf = str1.lastIndexOf("Kumar");

System.***out***.println("last index: " + lastIndexOf);

// 5. charAt() --> Used to find character at index.

**char** charAt = str1.charAt(5);

System.***out***.println(charAt);

// 6. trim() --> Used to remove whitespace from begging and end.

System.***out***.println(str1.trim());

// 7. isEmpty() --> Used to checks whether a String is empty or not

System.***out***.println("isEmpty check: " + str1.isEmpty());

// 8. equals() --> Used to check if two object content is same or not if same

// then return true or if not same then return false.

System.***out***.println(str1.equals(str2));

// 9. equalsToIgnoreCase() --> Used to check content of two object by ignoring

// case if content is same then return true if not same then reurn false..

System.***out***.println(str1.equalsIgnoreCase(str2));

// 10. startWith() --> Used to check string is start with given value or not.

System.***out***.println("startWith check: " + str1.startsWith("A"));

// 11. endsWith() --> Used to check string is ends with given string or not.

System.***out***.println("endsWith check:" + str1.endsWith("e"));

// 12. contains() --> Used to check value present in string or not.

System.***out***.println(str1.contains("singh"));

// 13. replace(old,new) --> Used to replace old value to new value.

System.***out***.println("Replacing old value with new value: " + str.replace("Amarjeet", "Sanjeet"));

// 14. replaceFirst(old,new) --> Used to replace first value

System.***out***.println(str.replaceFirst("Amarjeet", "Rajnish"));

// 15. replaceAll(old,new) --> Used to replace all value

System.***out***.println(str1.replaceAll("Amarjeet Kumar Singh", "Rajnish Kumar Singh"));

// 16. toLowerCase() --> Used to convert upper case to lower case.

System.***out***.println(str1.toLowerCase());

// 17. toUpperCase() --> Used to convert lower into upper case.

System.***out***.println(str1.toUpperCase());

// 18. join() --> Used to join two element with given delimeters.

System.***out***.println(String.*join*("---", str1, str2));

// 19. subSequence() --> Used to find the element between given index

System.***out***.println(str1.subSequence(3, 6));

// 20 substring() --> Used to find the element betwen given index.

System.***out***.println(str1.substring(3, 6));

// Note: substring() and subSequence() is working but syntax wise different.

}

}

**Explain StringBuffer class and its methods?**

StringBuffer class is used to create mutable string object StringBuffer class is similar to String class only difference is String class is immutable.

StringBuffer class is threat safe means synchronized and performace wise slow.

**Methods:**

                  // 1. append() --> Used to add value at last of the index it is like concat().

System.*out*.println(sb.append("From Bihar"));

// 2. insert() --> Used to insert the element at given position.

System.*out*.println(sb.insert(20, "Working at Edifecs Technologies"));

// 3. replace() -->

System.*out*.println(sb.replace(20, 62, "Working at UpWork India"));

// 4. delete() --> Used to delete element from position.

System.*out*.println(sb.delete(20, 62));

// 5. reverse() --> Used to reverse the element

System.*out*.println(sb.reverse());

//6. Capacity() 🡪 used to find the default initial capacity of StringBuffer class

System.out.println(sb.capacity()); //default capacity is 16

**StringBuilder class and its methods?**

StringBuilder class is used to create mutable string object StringBuilder class is similar to String class only difference is String class is immutable.

StringBuilder class is not threat safe means non-synchronized and performace wise fast

**Methods:**

                  // 1. append() --> Used to add value at last of the index it is like concat().

System.*out*.println(sb.append("From Bihar"));

// 2. insert() --> Used to insert the element at given position.

System.*out*.println(sb.insert(20, "Working at Edifecs Technologies"));

// 3. replace() -->

System.*out*.println(sb.replace(20, 62, "Working at UpWork India"));

// 4. delete() --> Used to delete element from position.

System.*out*.println(sb.delete(20, 62));

// 5. reverse() --> Used to reverse the element

System.*out*.println(sb.reverse());

//6. Capacity() 🡪 used to find the default initial capacity of StringBuffer class

System.out.println(sb.capacity()); //default capacity is 16

**17.What is the difference between String, StringBuffer and StringBuilder?**

|  |  |  |
| --- | --- | --- |
| **String** | **StringBuffer** | **StringBuilder** |
| The String objects are immutable. | The StringBuffer objects are mutable. | The StringBuilder object is mutable. |
| String objects stored in Heap area as well inside string constant pool. | StringBuffer objects are only stored inside the Heap area. | StringBuilder objects are only stored inside the Heap area. |
| String objects are synchronized, means  thread safe. | StringBuffer objects are synchronized, means  thread safe. | StringBuilder objects are non-synchronized, which means they are not thread safe. |
| Performance is slow. | Performance is slow**.** | Performance is fast. |

**Type Casting :**

Type casting means whenever you are assigning/converting one type of data to another type called type casting, There are two types of casting.

1. Widening or automatic casting→ **smaller into larger.**

**byte -> short -> char -> int -> long -> float -> double**

public class Main {

public static void main(String[] args) {

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt); // Outputs 9

System.out.println(myDouble); // Outputs 9.0

}

}

1. Narrowing or manual casting→ **larger into smaller.**

double -> float -> long -> int -> char -> short -> byte

public class Main {

public static void main(String[] args) {

double myDouble = 9.78d;

int myInt = (int) myDouble; // Manual casting: double to int

System.out.println(myDouble); // Outputs 9.78

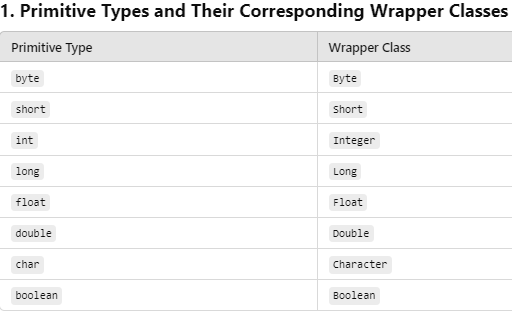
System.out.println(myInt); // Outputs 9

}

}

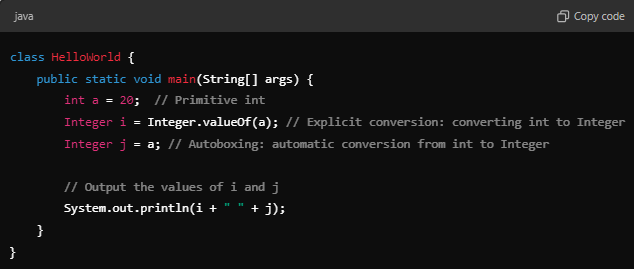
### **Wrapper Classes in Java:**

Wrapper classes in Java are used to convert primitive data types into objects. Each primitive type has a corresponding wrapper class, which allows us to use primitives in situations that require objects, such as in collections or when using generics.

****

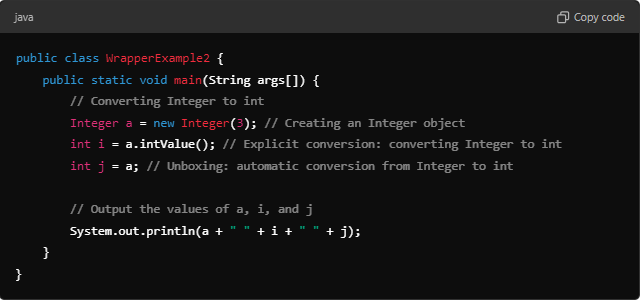
## **Autoboxing**

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short. **Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects**.



## **Unboxing:**

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives



### Object Class in Java

The Object class is the parent class of all classes in Java by default and is present in the java.lang package. It serves as the topmost class in the hierarchy, meaning all classes are derived from it. This class provides essential methods that can be overridden to define the behavior of objects.

### Methods of the Object Class

Here’s a brief explanation of some key methods of the Object class:

1. **hashCode()**
   * Returns an integer representation (hash code) of the object. This method is crucial for the proper functioning of hash-based collections like HashMap.
   * The default implementation returns a unique integer for each object, but it can be overridden to return a hash code based on the object's fields.
2. **toString()**
   * Returns a string representation of the object. The default implementation returns a string that consists of the class name followed by the object's hash code.
   * It is common to override this method to provide a meaningful description of the object.
3. **equals(Object obj)**
   * Compares this object with the specified object for equality. The default implementation checks for reference equality (i.e., whether they are the same object).
   * It is often overridden to define equality based on the object's state (i.e., field values).
4. **getClass()**
   * Returns the runtime class of the object. This method can be used to retrieve the class type of an object at runtime, which can be helpful for reflection.
5. **notify()**
   * Wakes up a single thread that is waiting on this object's monitor. This method is used in synchronization to manage thread communication.
   * It must be called from a synchronized context (inside a synchronized method or block).
6. **notifyAll()**
   * Wakes up all threads that are waiting on this object's monitor. Like notify(), it also requires a synchronized context.
   * It is used when you want to ensure that all waiting threads are notified to check the condition they are waiting for.
7. **finalize()**
   * This method is called by the garbage collector when the object is about to be reclaimed. It allows you to perform cleanup operations (like releasing resources) before the object is destroyed.
   * Note: finalize() has been deprecated in Java 9 and later due to its unpredictability.
8. **clone()**
   * Creates and returns a copy of the object. This method allows for creating a shallow copy of the object.
   * To use clone(), the class must implement the Cloneable interface, and the clone() method must be overridden to allow for cloning.
9. **wait()**
   * Causes the current thread to wait until another thread invokes the notify() or notifyAll() method on the same object. This method is also used in synchronization to manage thread communication.
   * Like notify(), it must be called from a synchronized context.

**Math Class:**

The Math class in Java is a utility class that provides a variety of mathematical operations such as logarithms, square roots, trigonometric functions, random number generation, and more. The methods in the Math class are static, meaning you do not need to create an instance of Math to use them.

Here's a complete guide to the Math class with code examples:

### 1. **Basic Methods**

#### Math.abs(): Absolute Value

This method returns the absolute value of a number (i.e., the positive version of the number).

public class MathExample {

public static void main(String[] args) {

int x = -10;

double y = -12.5;

System.out.println(Math.abs(x)); // Output: 10

System.out.println(Math.abs(y)); // Output: 12.5

}

}

#### Math.max() and Math.min(): Maximum and Minimum

These methods return the maximum or minimum of two numbers.

public class MathExample {

public static void main(String[] args) {

int a = 5, b = 10;

System.out.println(Math.max(a, b)); // Output: 10

System.out.println(Math.min(a, b)); // Output: 5

}

}

#### Math.sqrt(): Square Root

This method returns the square root of a number.

public class MathExample {

public static void main(String[] args) {

double number = 16;

System.out.println(Math.sqrt(number)); // Output: 4.0

}

}

#### Math.pow(): Power

This method raises the first argument to the power of the second argument (e.g., aba^bab).

public class MathExample {

public static void main(String[] args) {

double base = 2;

double exponent = 3;

System.out.println(Math.pow(base, exponent)); // Output: 8.0

}

}

#### Math.random(): Random Numbers

This method returns a pseudo-random number between 0.0 (inclusive) and 1.0 (exclusive).

public class MathExample {

public static void main(String[] args) {

double randomValue = Math.random();

System.out.println(randomValue); // Output: Some random value between 0 and 1

}

}

You can scale the random value to a desired range:

public class MathExample {

public static void main(String[] args) {

int min = 1, max = 100;

int randomNumber = (int)(Math.random() \* (max - min + 1) + min);

System.out.println(randomNumber); // Output: Random number between 1 and 100

}

}

### 2. **Trigonometric Methods**

#### Math.sin(), Math.cos(), Math.tan(): Sine, Cosine, and Tangent

These methods return the sine, cosine, or tangent of an angle (in radians).

public class MathExample {

public static void main(String[] args) {

double angle = Math.toRadians(45); // Convert degrees to radians

System.out.println(Math.sin(angle)); // Output: 0.7071067811865476 (sin 45°)

System.out.println(Math.cos(angle)); // Output: 0.7071067811865476 (cos 45°)

System.out.println(Math.tan(angle)); // Output: 1.0 (tan 45°)

}

}

#### Math.toRadians() and Math.toDegrees(): Convert Angle Units

These methods are used to convert angles between degrees and radians.

public class MathExample {

public static void main(String[] args) {

double degrees = 180;

double radians = Math.toRadians(degrees);

System.out.println(radians); // Output: 3.141592653589793 (π radians)

double rad = Math.PI;

System.out.println(Math.toDegrees(rad)); // Output: 180.0 degrees

}

}

#### Math.asin(), Math.acos(), Math.atan(): Inverse Trigonometric Functions

These methods return the inverse sine, cosine, or tangent of a value.

public class MathExample {

public static void main(String[] args) {

double value = 0.5;

System.out.println(Math.asin(value)); // Output: 0.5235987755982989 (in radians)

System.out.println(Math.acos(value)); // Output: 1.0471975511965979 (in radians)

System.out.println(Math.atan(value)); // Output: 0.4636476090008061 (in radians)

}

}

### 3. **Exponential and Logarithmic Methods**

#### Math.exp(): Exponent

This method returns the exponential of a number, i.e., exe^xex where eee is Euler's number (approximately 2.71828).

java

Copy code

public class MathExample {

public static void main(String[] args) {

double value = 2;

System.out.println(Math.exp(value)); // Output: 7.38905609893065 (e^2)

}

}

#### Math.log(): Natural Logarithm

This method returns the natural logarithm (base eee) of a number.

public class MathExample {

public static void main(String[] args) {

double value = 7.389056;

System.out.println(Math.log(value)); // Output: 2.0

}

}

#### Math.log10(): Logarithm Base 10

This method returns the logarithm of a number with base 10.

public class MathExample {

public static void main(String[] args) {

double value = 100;

System.out.println(Math.log10(value)); // Output: 2.0

}

}

### 4. **Rounding Methods**

#### Math.round(): Round to Nearest Integer

This method rounds a floating-point number to the nearest integer.

public class MathExample {

public static void main(String[] args) {

double value = 5.67;

System.out.println(Math.round(value)); // Output: 6

}

}

#### Math.ceil(): Round Up

This method returns the smallest integer greater than or equal to the argument.

public class MathExample {

public static void main(String[] args) {

double value = 5.01;

System.out.println(Math.ceil(value)); // Output: 6.0

}

}

#### Math.floor(): Round Down

This method returns the largest integer less than or equal to the argument.

public class MathExample {

public static void main(String[] args) {

double value = 5.99;

System.out.println(Math.floor(value)); // Output: 5.0

}

}

### 5. **Other Utility Methods**

#### Math.signum(): Sign of a Number

This method returns -1.0, 0.0, or 1.0 depending on whether the argument is negative, zero, or positive.

public class MathExample {

public static void main(String[] args) {

System.out.println(Math.signum(-10)); // Output: -1.0

System.out.println(Math.signum(0)); // Output: 0.0

System.out.println(Math.signum(10)); // Output: 1.0

}

}

#### Math.hypot(): Hypotenuse Calculation

This method returns the length of the hypotenuse of a right-angled triangle given the lengths of the other two sides.

public class MathExample {

public static void main(String[] args) {

double a = 3;

double b = 4;

System.out.println(Math.hypot(a, b)); // Output: 5.0

}

}

### 6. **Math Constants**

#### Math.PI: Pi Constant

This constant represents the value of π (approximately 3.14159).

#### Math.E: Euler's Number

This constant represents the value of eee (approximately 2.71828).

public class MathExample {

public static void main(String[] args) {

System.out.println(Math.PI); // Output: 3.141592653589793

System.out.println(Math.E); // Output: 2.718281828459045

}

}

### Summary

* **Common Methods**: abs(), max(), min(), sqrt(), pow(), random().
* **Trigonometry**: sin(), cos(), tan(), asin(), acos(), atan(), toRadians(), toDegrees().
* **Exponential/Logarithmic**: exp(), log(), log10().
* **Rounding**: round(), ceil(), floor().
* **Other**: signum(), hypot(), and constants like PI and E.

**Java Shell Tool (JShell)**

**JShell** is an interactive command-line tool introduced in **Java 9** that enables users to execute Java code directly from the shell, displaying output immediately. It functions as a **REPL** (Read-Evaluate-Print Loop) environment, allowing developers to evaluate and test code snippets quickly without creating a full Java program structure. JShell simplifies and speeds up Java development, especially useful for quick tests and prototyping.

**Key Features of JShell**

* **Interactive Execution**: Allows immediate feedback on code, making it perfect for quick experimentation and testing.
* **Code Evaluation**: JShell can evaluate individual expressions, statements, methods, and classes, providing flexibility in code testing.
* **Command-Line Interface**: Run from the terminal or command prompt, making it accessible and lightweight.
* **Direct Output**: Displays results instantly, making debugging and learning Java easier and faster.

**Advantages of Using JShell**

JShell significantly reduces the steps required to write, compile, and run Java code. With JShell, you can skip many of the traditional steps involved in Java development:

1. **Simplified Development**: In a typical Java development workflow:
   * Open an editor and write the program.
   * Save the program to a file.
   * Compile the program.
   * Edit if any compile-time error occurs.
   * Run the program.
   * Edit if any runtime error occurs and repeat.

With JShell, you can skip all these steps. You can simply enter your Java code directly in the shell, and it will be executed immediately, showing output right away.

1. **No Need for Boilerplate Code**:
   * You can test small snippets or business logic without creating a full Java class with a main method.
   * For example, you can write a simple "Hello, World!" program without creating a class or method.
2. **Efficient Testing and Debugging**:
   * Allows for quick testing of statements, expressions, methods, or even small classes without the need to manage a file structure.
   * You can make changes on the fly and re-evaluate code snippets without leaving the shell.
3. **Ideal for Learning and Prototyping**:
   * Provides a friendly environment for beginners to learn Java syntax interactively.
   * Helps in testing out new APIs or Java features without setting up an entire project.

**Practical Example in JShell**

To understand the efficiency of JShell, compare the standard Java setup process with JShell's simplicity.

1. **Without JShell**:
   * Write code in an editor (e.g., a "Hello, World!" program).
   * Save the file as HelloWorld.java.
   * Compile using javac HelloWorld.java.
   * Run the program with java HelloWorld.
2. **With JShell**:
   * Open JShell by typing jshell in the terminal.
   * Write the statement directly:

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

* + JShell immediately displays the output:

Hello, World!

**Summary**

JShell simplifies the process of writing, compiling, and running Java code by providing an interactive shell that allows for immediate execution. This tool removes the need for repetitive steps in program creation, enabling developers to test ideas, learn, and debug faster and more effectively.

**Practical of Jshell**

**1. Starting and Exiting JShell**

* **Starting JShell:**

jshell

* + Type jshell in the terminal or command prompt to start JShell.
* **Exiting JShell:**

/exit

* + Type /exit to leave JShell.

**2. Basic Commands**

* **Executing Expressions:**  
  JShell evaluates expressions directly.

jshell> 2 + 3

Output:

$1 ==> 5

* **Printing Output:**  
  JShell prints the result automatically, but you can also use System.out.println.

jshell> System.out.println("Hello, JShell!")

Output:

Hello, JShell!

* **Assigning Variables:**

jshell> int a = 10;

jshell> String name = "JShell";

**3. Commands in JShell**

JShell provides several commands, all of which start with /.

* **Listing Commands:**

/help

* + Shows a list of all commands available in JShell.
* **Viewing All Variables, Methods, and Imports:**

/vars // Lists all defined variables.

/methods // Lists all defined methods.

/imports // Lists all imports in the current session.

/list // Lists of sources you have type so far

* **Editing Code:**

/edit <variable\_name> // Opens the editor for the specified variable or method.

* **Resetting JShell:**

/reset

* + Clears all defined variables, methods, and imports.
* **History of Commands:**

/history

* + Shows the command history in the session.
* **Loading and Saving JShell Code:**

/save mySession.jsh

/open mySession.jsh

* + /save saves the current JShell session to a file.
  + /open loads a saved session.

**4. Working with Methods and Classes**

* **Defining Methods:**

jshell> int add(int x, int y) {

return x + y;

}

jshell> add(5, 7)

Output:

$3 ==> 12

* **Creating Classes:**

jshell> class Person {

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

void display() {

System.out.println(name + " is " + age + " years old.");

}

}

jshell> Person p = new Person("Alice", 30);

jshell> p.display();

Output:

Alice is 30 years old.

* **Creating Interfaces:**

jshell> interface Greetable {

void greet();

}

jshell> class Greeter implements Greetable {

public void greet() {

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

}

}

jshell> Greeter greeter = new Greeter();

jshell> greeter.greet();

Output:

Hello from JShell!

**5. Importing Packages**

JShell has a few default imports (like java.util.\*, java.io.\*). You can import additional packages as needed.

jshell> import java.time.LocalDate;

jshell> LocalDate.now();

Output:

$7 ==> 2024-10-27

**6. JShell Snippets**

Each executed line is stored as a snippet and is identified by $<number>. You can re-evaluate or use previous results:

jshell> int x = 5;

jshell> int y = x \* 2;

jshell> int result = $3 + 10; // Using previous snippet result

**7. Using /set Command for Configuration**

JShell allows setting preferences with the /set command.

* **Set Feedback Mode:**

/set feedback verbose // Provides detailed output

/set feedback concise // Provides minimal output

* **Setting Prompt:**

/set prompt "jshell> " // Custom prompt

**Unit-3 Notes**

**Oops in Java**

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.

**Constructor in Java**

* 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.

**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

**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();

}

}

**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).

**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

**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.

**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.

**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**, and **private method we can write since java9**, 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.