**Java Notes By me**

**Day 1:**

1. **Features of Java.**

**Ans :**

**1. Object-Oriented Programming (OOP).**

* Description: Java is an object-oriented language, meaning everything is represented as objects with attributes (fields) and behaviors (methods).
* Real-Time Example: Consider a banking system:
  + You have a class Account with attributes like balance, accountNumber, and methods like deposit(), withdraw().
  + Each customer can have their own account object with specific details and actions.

**2. Platform Independence (Write Once, Run Anywhere).**

* Description: Java code is compiled into bytecode, which can run on any machine with a Java Virtual Machine (JVM), making Java platform-independent.
* Real-Time Example: You can write a Java application (e.g., a weather application) on a Windows machine and run it seamlessly on Linux or macOS without modification.
* Interview Tip: Emphasize how JVM acts as an intermediary between Java code and the machine's operating system, making it adaptable across various platforms.

**3.** **Automatic Garbage Collection.**

* Description: Java manages memory allocation and deallocation automatically. The garbage collector periodically frees up memory by removing objects that are no longer in use.
* Real-Time Example: If you are working with an e-commerce application, after a user session ends, the objects associated with the session, such as their cart items, are automatically cleaned up to free memory.
* Interview Tip: Explain how the garbage collector helps reduce memory leaks and improves application performance.

**4. Multithreading.**

* Description: Java provides built-in support for multithreading, allowing the execution of multiple threads concurrently, which is useful for performing tasks in parallel.
* Real-Time Example: In a live streaming application, one thread could handle video rendering, while another handles user input, and a third one streams the content. This ensures smooth performance.
* Interview Tip: Be ready to explain how threads work in Java and how it helps in building responsive applications.

**5. Rich API (Application Programming Interface).**

* Description: Java provides a vast set of libraries that offer functionalities for I/O, networking, data structures, utilities, etc.
* Real-Time Example: Java's java.util package provides classes like ArrayList, HashMap, and HashSet, which can be used in a system to store user data or manage product information.
* Interview Tip: Highlight how Java's extensive library ecosystem helps developers build robust applications quickly.

**6. Security Features.**

* Description: Java provides several security features like bytecode verification, secure class loading, and cryptography APIs.
* Real-Time Example: In an online payment system, Java's security mechanisms can be used to encrypt sensitive data like credit card numbers during transaction processing.
* Interview Tip: Mention how the Java security manager, along with APIs for encryption and authentication, ensures data integrity and prevents unauthorized access.

**7. Exception Handling.**

* Description: Java provides a powerful mechanism for handling runtime errors using try, catch, finally, and custom exceptions, allowing for robust error management.
* Real-Time Example: In an online booking system, if a user tries to book a flight that is no longer available, an exception can be thrown, and the system can gracefully handle it by informing the user.
* Interview Tip: Be sure to explain how exception handling allows for better user experiences and helps ensure systems continue running smoothly despite errors.

**8. Portability.**

* Description: The "Write Once, Run Anywhere" philosophy ensures that Java applications can run on any device or platform that supports JVM, which is widely adopted.
* Real-Time Example: A mobile app built in Java for Android can be ported to different Android devices with different screen sizes, operating systems, and configurations.
* Interview Tip: Discuss how Java achieves portability through the use of bytecode and JVM.

**9. Network-Centric Programming.**

* Description: Java offers a rich set of APIs for building networked applications like HTTP-based web services, socket programming, and remote method invocation (RMI).
* Real-Time Example: A chat application uses Java's socket programming to establish a network connection between the server and multiple clients, enabling real-time communication.
* Interview Tip: Focus on Java's built-in networking capabilities like java.net for tasks such as sending HTTP requests or building custom protocols.

**10. Database Connectivity (JDBC).**

* Description: Java Database Connectivity (JDBC) allows Java applications to interact with databases (e.g., MySQL, Oracle) for performing CRUD (Create, Read, Update, Delete) operations.
* Real-Time Example: A customer relationship management (CRM) system uses JDBC to interact with a backend database, storing and retrieving customer data.
* Interview Tip: Be prepared to explain how JDBC establishes a connection, executes queries, and handles results with real-time examples.

**11. JavaFX and Swing (GUI Development).**

* Description: Java offers libraries like JavaFX and Swing for building graphical user interfaces (GUIs) in desktop applications.
* Real-Time Example: A point-of-sale system can use JavaFX to provide a user-friendly interface for salespeople to quickly process transactions.
* Interview Tip: If the interview is about desktop application development, mention how JavaFX and Swing enable the creation of rich user interfaces.

**12. Distributed Computing (RMI and EJB).**

* Description: Java provides tools for developing distributed applications where components communicate over a network.
* Real-Time Example: A stock trading platform uses Remote Method Invocation (RMI) to allow clients to remotely access trading services on a server.
* Interview Tip: Discuss Java’s Enterprise JavaBeans (EJB) and RMI for building scalable, distributed applications.

**13. Strong Memory Management.**

* Description: Java provides strong memory management through automatic garbage collection, which helps prevent memory leaks.
* Real-Time Example: In a game development scenario, Java automatically manages memory when objects like game levels, characters, and scenes are created and discarded as needed.
* Interview Tip: Be ready to explain how Java uses heap memory for object allocation and stack memory for method execution, and how this supports automatic memory management.

**14. Simple Syntax.**

* Description: Java syntax is simple, easy to learn, and similar to C, making it a good choice for developers transitioning from other programming languages.
* Real-Time Example: In a school management system, developers can quickly code features like student registration or grading using Java’s simple and clean syntax.

1. **Features of JDK 1.8.**

**Ans : 1. Lambda Expressions**

* **Description:** Lambda expressions allow you to write concise, expressive code by enabling you to treat functionality as a method argument or create a method that can be passed around. A lambda expression is essentially an anonymous function that can be used to implement a method defined by a functional interface.
* Benefit: Reduces boilerplate code and makes the code more readable, especially when working with functional programming patterns like map, filter, and reduce.

**2. Functional Interfaces**

* Description: A functional interface is an interface with exactly one abstract method, and it may contain multiple default or static methods. Lambda expressions can be used to instantiate functional interfaces. Common examples include interfaces like Runnable, Callable, Comparator, and Predicate.
* Benefit: Enables functional programming features in Java, and allows lambda expressions to be used in a type-safe manner.

**3. Streams API**

* Description: The Streams API allows you to process sequences of elements (such as collections) in a functional style. It supports operations like filtering, mapping, and reducing, allowing for more concise, readable, and parallelizable code. The API provides a high-level abstraction for working with data sequences.
* Benefit: Facilitates the processing of large amounts of data in a declarative and readable way, and supports parallelism to improve performance on multi-core processors.

**4. Default Methods in Interfaces**

* Description: Java 8 introduced the ability to define methods with default implementations in interfaces using the default keyword. This allows new methods to be added to interfaces without breaking existing implementations.
* Benefit: Supports backward compatibility with existing code, making it easier to add new functionality to interfaces without requiring changes to implementing classes.

**5. Method References.**

* Description: Method references provide a shorthand way to call methods directly, replacing more verbose lambda expressions. They enable referring to methods by their names in a clear and concise manner.
* Benefit: Increases code clarity and conciseness when you need to pass a method as a parameter, reducing boilerplate code.

**6. New Date and Time API (java.time).**

* Description: Java 8 introduced a new Date and Time API (java.time package) that addresses the shortcomings of the older java.util.Date and java.util.Calendar classes. The new API is immutable and thread-safe, offering much better handling of dates, times, and durations.
* Benefit: Provides a more intuitive and precise way to handle date and time calculations, comparisons, and formatting, and eliminates the issues of Date and Calendar classes.

**7. Optional Class.**

* Description: The Optional class is a container object used to represent the presence or absence of a value. It is often used to prevent NullPointerException by providing methods that handle missing values in a more elegant and readable way.
* Benefit: Helps avoid null-related errors and improves code quality by encouraging developers to explicitly handle the absence of values.

**8. Nashorn JavaScript Engine**

* Description: Java 8 introduced the Nashorn JavaScript engine, which allows Java applications to run JavaScript code directly. It replaces the older Rhino engine and provides faster execution of JavaScript.
* Benefit: Enhances the ability to integrate Java with JavaScript, enabling developers to execute JavaScript code within Java applications, often used in web development or dynamic scripting.

**9. New Collection Methods.**

* Description: Java 8 added several new methods to the Collection interface, such as forEach(), removeIf(), replaceAll(), and sort(). These methods make it easier to work with collections in a more functional and expressive way.
* Benefit: Provides a more intuitive and concise way to manipulate collections without needing to write verbose loops or conditional checks.

**10. Parallel Streams.**

* Description: Java 8 allows you to process data in parallel by converting a regular stream into a parallel stream using the .parallelStream() method. This enables efficient use of multiple processor cores, allowing for parallel processing of data.
* Benefit: Helps improve performance when working with large datasets by utilizing multi-core processors without the need to manage threads manually.

**11. Collectors API.**

* Description: The Collectors utility class provides various predefined methods to collect data from streams, such as toList(), toSet(), joining(), and groupingBy(). These collectors can be used to gather results from stream operations into collections or other types of results.
* Benefit: Simplifies the process of collecting and transforming data from streams into different formats, such as lists, sets, or maps, without having to write custom collection logic.

**12. CompletableFuture.**

* Description: CompletableFuture is a class introduced in Java 8 that facilitates asynchronous programming by allowing you to write non-blocking code. It provides methods for completing tasks asynchronously and for composing multiple asynchronous tasks.
* Benefit: Simplifies asynchronous programming and makes it easier to handle multiple tasks running concurrently, providing better support for non-blocking I/O operations and concurrent workflows.

1. **Features of java 15.**

**Ans :**

**1. Sealed Classes (Preview) [JEP 360]**

Sealed classes provide a way to control which other classes or interfaces can extend or implement them. This helps in designing a well-defined class hierarchy and ensures that only approved subclasses can be created. The primary goal of this feature is to enhance encapsulation and maintainability in object-oriented designs.

* Sealed classes restrict unauthorized subclassing, preventing unintended modifications to a class hierarchy.
* This feature improves security by enforcing explicit permissions for subclassing.
* It is particularly useful in domain-driven design where specific relationships between classes need to be strictly maintained.
* Permitted subclasses must be explicitly defined, improving readability and structure.

**2. Text Blocks (Standard Feature) [JEP 378]**

Text blocks provide a multi-line string literal feature, eliminating the need for escape sequences and making the representation of large text-based content much cleaner.

* This feature simplifies working with JSON, HTML, SQL queries, and multi-line strings, which previously required cumbersome escape sequences.
* Improves readability and reduces errors in large string literals.
* Automatically formats text while preserving the intended structure.
* Enhances developer productivity by reducing unnecessary string concatenation.

**3. Hidden Classes [JEP 371]**

Hidden classes are dynamically generated classes that are not discoverable by other Java code, making them useful for frameworks and runtime-generated proxies.

* These classes do not appear in standard class loaders, making them secure and efficient for temporary use.
* Primarily intended for JVM-based frameworks that create and manage classes dynamically at runtime.
* Helps reduce memory footprint as these classes do not remain in memory indefinitely.
* Useful in dynamic language implementations (such as Groovy, Kotlin, and Java proxy classes) where runtime-generated classes are frequently used.

1. **Features of jdk 17 .**

**Ans : Java Development Kit (JDK) 17, released in September 2021, is a Long-Term Support (LTS) version, meaning it will receive extended updates and support. This version introduced several significant enhancements in performance, security, and developer productivity.**

**1. Sealed Classes (Finalized) [JEP 409]**

Sealed classes were first introduced as a preview feature in Java 15 and are now fully integrated in JDK 17.

* They allow developers to control which classes or interfaces can extend them, preventing unwanted subclassing.
* This feature enhances code safety, maintainability, and readability by ensuring a controlled hierarchy.
* Sealed classes improve domain-driven design by enforcing specific subclass relationships, reducing errors in inheritance structures.

**2. Pattern Matching for Switch (Preview) [JEP 406]**

This feature enhances the switch statement by allowing it to match patterns in a more concise and readable manner.

* It simplifies handling different data types in switch cases, making code more expressive and reducing boilerplate.
* It brings a more functional programming style to Java, improving readability and maintainability.
* This is especially useful in data-driven applications, reducing the need for repetitive type checks and casting.

**3. Strongly Encapsulated JDK Internals [JEP 403]**

Java 17 strictly encapsulates internal APIs, reducing reliance on non-standard Java classes.

* Many internal APIs (such as sun.misc.Unsafe) are now completely restricted unless explicitly allowed.
* This change improves security, stability, and compatibility by preventing accidental usage of internal classes.
* Developers are encouraged to use official Java APIs instead of depending on internal, unsupported classes.

**4. Deprecation and Removal of Legacy Features.**

Java 17 removes and deprecates several outdated features to improve efficiency and modernize the language.

* Deprecation of Security Manager: The Security Manager, a legacy security feature, is deprecated due to limited real-world usage and better modern alternatives.
* Removal of RMI Activation System: The Remote Method Invocation (RMI) Activation System is removed as it was rarely used in modern applications.
* Removal of the Experimental AOT and JIT Compiler: These experimental features were removed due to low adoption rates and high maintenance costs.

1. **Features of JDk 21.**

**Ans :**

Java Development Kit (JDK) 21, released in **September 2023**, is a significant **Long-Term Support (LTS)** release that introduces several notable enhancements aimed at improving developer productivity, application performance, and code readability. Here are five key features of JDK 21:

**1. Virtual Threads [JEP 444]**

**Virtual Threads** revolutionize Java's concurrency model by allowing the creation of lightweight threads that are managed by the Java runtime rather than the operating system. This enhancement enables applications to scale more efficiently, supporting a higher number of concurrent tasks without the overhead associated with traditional platform threads.

[azul.com](https://www.azul.com/blog/jdk-21-delivers-virtual-threads-other-new-features-and-long-term-support/?utm_source=chatgpt.com)

**Key Benefits:**

* **Enhanced Scalability:** Supports a large number of concurrent operations, making it ideal for applications with numerous simultaneous connections.
* **Simplified Concurrency:** Allows developers to use a thread-per-task model without incurring significant performance penalties.

**2. Sequenced Collections [JEP 431]**

The introduction of **Sequenced Collections** addresses inconsistencies in the Java Collections Framework regarding the encounter order of elements. This feature adds new interfaces—SequencedCollection, SequencedSet, and SequencedMap—which define collections with a well-defined sequence, enhancing code predictability and reliability.

[baeldung.com](https://www.baeldung.com/java-lts-21-new-features?utm_source=chatgpt.com)

**Key Benefits:**

* **Consistent Encounter Order:** Ensures that collections maintain a predictable order of elements, reducing potential bugs related to unordered data.
* **Enhanced API Integration:** Provides a unified approach to handling ordered collections, simplifying development and maintenance.

**3. Record Patterns [JEP 440]**

**Record Patterns** extend pattern matching capabilities in Java, allowing for more expressive and concise data retrieval from record types. This feature simplifies the deconstruction of records, enabling developers to extract and process data more efficiently.

[shaaf.dev](https://shaaf.dev/post/2023-09-21-whats-new-developers-jdk-21/?utm_source=chatgpt.com)

**Key Benefits:**

* **Improved Data Manipulation:** Facilitates straightforward extraction of components from records, enhancing code clarity.
* **Seamless Integration:** Works cohesively with existing pattern matching features, promoting a more declarative programming style.

**4. String Templates (Preview) [JEP 430]**

Introduced as a preview feature, **String Templates** provide a safer and more readable approach to constructing strings that include dynamic expressions. This feature reduces common errors associated with string concatenation and enhances code maintainability.

[shaaf.dev](https://shaaf.dev/post/2023-09-21-whats-new-developers-jdk-21/?utm_source=chatgpt.com)

**Key Benefits:**

* **Enhanced Readability:** Allows embedding expressions directly within string literals, making code more intuitive.
* **Improved Security:** Mitigates risks such as injection attacks by handling variable interpolation more securely.

**5. Structured Concurrency (Preview) [JEP 453]**

**Structured Concurrency** aims to simplify concurrent programming by treating multiple tasks running in different threads as a single unit of work. This approach streamlines error handling and cancellation, improving the reliability and observability of concurrent code.

[oracle.com](https://www.oracle.com/java/technologies/javase/21-relnote-issues.html?utm_source=chatgpt.com)

**Key Benefits:**

* **Simplified Thread Management:** Groups related tasks, ensuring they complete together, which simplifies synchronization and error handling.
* **Enhanced Reliability:** Reduces the likelihood of thread leaks and other concurrency-related issues by managing task lifecycles collectively.

1. **Features of java 11.**

**Ans :**

Java 11, released in **September 2018**, is a **Long-Term Support (LTS)** version that introduced various enhancements to improve **performance, security, and developer productivity**. It also removed outdated APIs and incorporated modern features to streamline Java development.

**1. New String Methods**

Java 11 introduced several new methods in the String class to improve string manipulation, making code more readable and concise.

* **isBlank()** – Checks if a string is empty or contains only white spaces.
* **lines()** – Converts a multi-line string into a stream of individual lines.
* **strip(), stripLeading(), stripTrailing()** – Removes white spaces more efficiently than trim().
* **repeat(int count)** – Repeats a string multiple times.

These methods enhance efficiency when working with text data.

**2. HTTP Client (Standardized) [JEP 321]**

The **new HTTP Client API** was introduced as an official standard in Java 11. It supports both **synchronous and asynchronous** communication and provides built-in support for handling HTTP/2 requests.

**Key Benefits:**

* **Improved Performance**: Uses HTTP/2, which enhances speed and reduces network latency.
* **Simplified API**: More intuitive than the old HttpURLConnection.
* **Built-in WebSocket Support**: Enables real-time, bi-directional communication.

**3. Local-Variable Syntax for Lambda Parameters [JEP 323]**

Java 11 allows the use of the var keyword in **lambda expressions**, making code more readable and uniform.

**Key Benefits:**

* **Enhances Readability**: Developers can use the var keyword to make lambda expressions more concise.
* **Improves Code Consistency**: Allows uniformity when using var in regular variable declarations and lambda expressions.

**4. Removal of Java EE and CORBA Modules**

Java 11 **removes outdated and deprecated Java EE (Jakarta EE) and CORBA (Common Object Request Broker Architecture) modules**, which were rarely used in modern applications.

**Key Benefits:**

* **Reduces Memory Footprint**: By eliminating unused features, Java becomes lighter and faster.
* **Encourages Modern Alternatives**: Developers are encouraged to use up-to-date frameworks such as **Spring** or **Jakarta EE**.

**5. Flight Recorder and Mission Control [JEP 328]**

The **Java Flight Recorder (JFR)** and **Mission Control** tools, previously commercial features, were made freely available in Java 11.

**Key Benefits:**

* **Real-time Performance Monitoring**: Helps developers analyze CPU usage, memory allocation, and garbage collection.
* **Low Overhead**: Can be used in production without affecting application performance.

**6. Z Garbage Collector (ZGC) [JEP 333]**

Java 11 introduced the **Z Garbage Collector (ZGC)**, an experimental **low-latency** garbage collector designed for applications requiring minimal GC pause times.

**Key Benefits:**

* **Extremely Low Pause Times**: Maintains response times below **10ms**, even for large heap sizes.
* **Scalability**: Supports heap sizes ranging from **a few GB to multiple TBs**.
* **Efficient Performance**: Improves application responsiveness in real-time and large-scale systems.

**7. Epsilon Garbage Collector [JEP 318]**

Java 11 introduced **Epsilon GC**, a garbage collector that **does nothing**—it only allocates memory but never frees it.

**Key Benefits:**

* **Useful for Performance Testing**: Helps benchmark memory allocation without GC interference.
* **Low Overhead**: Avoids GC-related CPU costs in scenarios where memory management is handled externally.

**8. Running Java Files without Compilation**

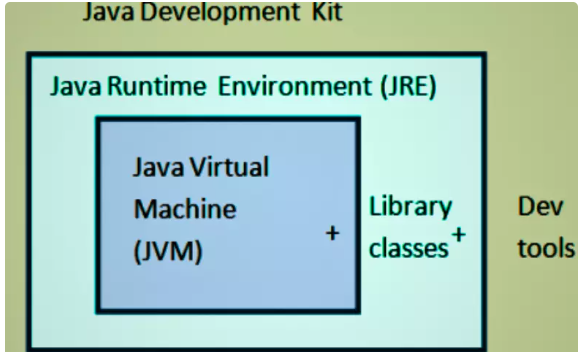
Java 11 allows developers to run **single-file Java programs without explicitly compiling them** using javac.

**Key Benefits:**

* **Faster Development**: Enables quick scripting without creating a full project structure.
* **Improves Productivity**: Reduces boilerplate steps for small scripts and testing.

1. **Working of JDK.**

**Ans:**



**Working of JDK (Java Development Kit) in Detail**

**1.** **Writing Code**

* Java programs are written in a .java file using any text editor or an Integrated Development Environment (IDE) like IntelliJ IDEA, Eclipse, or VS Code.
* The code follows Java syntax and is written inside classes and methods.

2. **Compilation Process**

* The Java compiler (javac) compiles the .java file into a .class file, which contains bytecode.
* Bytecode is a platform-independent intermediate code that can be executed on any system with a Java Virtual Machine (JVM).

3. **Class Loading (JVM Process Begins)**

* When a Java program runs, the ClassLoader loads the required .class files into memory.
* It includes three main class loaders:
  + Bootstrap ClassLoader – Loads core Java classes from the JDK.
  + Extension ClassLoader – Loads additional Java libraries.
  + Application ClassLoader – Loads user-defined classes from the program.

4. **Bytecode Verification**

* The Bytecode Verifier checks the .class file to ensure that it follows Java security and syntax rules.
* This step prevents issues like illegal memory access or stack overflows.

5. **Execution (Interpretation & Compilation by JVM)**

* The Java Interpreter inside the JVM reads and translates bytecode into machine code for execution.
* JVM can use Just-In-Time (JIT) Compilation to convert frequently used bytecode into native machine code for faster performance.

6. **Runtime Environment (JRE)**

* The Java Runtime Environment (JRE) provides essential Java libraries and runtime support.
* It includes Java APIs, garbage collection, memory management, and exception handling.

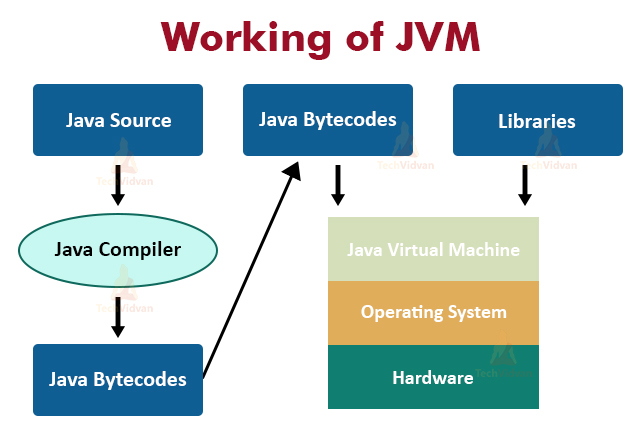
7. **Program Execution and Output**

* After processing the bytecode, the CPU executes the translated machine code, and the program produces the expected output.
* The execution may involve multiple Java features like multithreading, exception handling, and memory management (garbage collection).

**Summary of Working Steps**

1. Write Java code in a .java file.
2. Compile the code using javac to generate bytecode (.class file).
3. Load the class into memory using the ClassLoader.
4. Verify the bytecode for security and correctness.
5. Interpret and execute the bytecode using JVM and JIT Compilation.
6. Provide runtime support through JRE for memory management and execution.
7. Display the final output of the Java program.
8. **Working of JVM.**

**Ans :**

****

**Working of JVM (Java Virtual Machine) in Detail**

The Java Virtual Machine (JVM) is an essential part of the Java Runtime Environment (JRE) that executes Java bytecode on different platforms, making Java a platform-independent language. It converts compiled Java bytecode into native machine code for execution.

**1. Components of JVM**

**1.1 Class Loader Subsystem**

* The ClassLoader loads Java classes into memory when required.
* It includes three types of class loaders:
  1. Bootstrap ClassLoader – Loads core Java classes (java.lang.\*, java.util.\*, etc.).
  2. Extension ClassLoader – Loads additional Java libraries from the ext directory.
  3. Application ClassLoader – Loads user-defined classes from the project folder.

**1.2 Memory Areas of JVM**

The JVM manages memory using different memory areas:

1. Method Area
   * Stores class metadata, static variables, and method code.
2. Heap Area
   * Stores objects and instance variables.
   * Managed by Garbage Collection (GC).
3. Stack Area
   * Stores method call details like local variables and partial results.
   * Each thread has its own stack.
4. PC Register (Program Counter Register)
   * Holds the memory address of the currently executing instruction.
5. Native Method Stack
   * Stores native method (non-Java) calls used in the program.

**1.3 Execution Engine**

The Execution Engine converts bytecode into machine code and executes it. It consists of:

1. Interpreter
   * Reads and executes bytecode line by line.
   * Slower than JIT compilation.
2. Just-In-Time (JIT) Compiler
   * Converts frequently used bytecode into native code for faster execution.
   * Improves performance significantly.
3. Garbage Collector (GC)
   * Automatically deallocates unused memory from the heap.
   * Uses different algorithms like Mark and Sweep, Generational GC, etc.

**1.4 Java Native Interface (JNI) and Java Native Method Interface (JNMI)**

* Allows JVM to interact with native (non-Java) code like C/C++ for system-level operations.
* JNI enables Java programs to call platform-specific libraries.

**2. Working of JVM (Step-by-Step)**

**Step 1:** Class Loading

* The JVM loads .class files into memory using the ClassLoader.
* These files contain bytecode, which is platform-independent.

**Step 2:** Bytecode Verification

* The Bytecode Verifier checks the loaded bytecode to ensure it follows Java security rules.
* Prevents unauthorized memory access or invalid bytecode execution.

**Step 3**: Memory Allocation

* JVM allocates memory in different areas (Method Area, Heap, Stack, etc.).
* Objects are stored in the Heap, and method calls are handled in the Stack.

**Step 4:** Execution by the Execution Engine

* The Execution Engine reads bytecode and executes it using the Interpreter or JIT Compiler.
* The JIT Compiler improves performance by converting bytecode into machine code when necessary.

**Step 5:** Garbage Collection

* JVM automatically removes unused objects from memory to optimize performance.
* Garbage Collectors manage heap space efficiently.

**Step 6:** Output Generation

* After execution, the final output is displayed to the user.

**3. Summary of JVM Working Process**

1. Loads Java bytecode into memory using ClassLoader.
2. Verifies bytecode for security and correctness.
3. Allocates memory for program execution.
4. Interprets or compiles bytecode into machine code.
5. Executes the program using CPU.
6. Performs garbage collection to free memory.
7. Generates final output of the Java program.

9.**Working of JRE.**

**Ans: Working of JRE (Java Runtime Environment) in Detail**

**1. Introduction to JRE**

The Java Runtime Environment (JRE) is a part of the Java Development Kit (JDK) that provides the necessary environment for running Java applications. It does not include the Java compiler (javac), so it cannot be used for Java development. However, it contains the Java Virtual Machine (JVM) and essential libraries to execute Java programs.

2**. Components of JRE**

2**.1 Java Virtual Machine (JVM)**

* The JVM is responsible for executing Java bytecode.
* It translates bytecode into machine code using an Interpreter or Just-In-Time (JIT) Compiler.

**2.2 Java Class Libraries**

* JRE includes a set of predefined Java libraries (APIs) required for program execution.
* These libraries include packages like:
  + java.lang (Core Java classes)
  + java.util (Utility functions)
  + java.io (Input/Output operations)
  + java.net (Networking support)
  + java.sql (Database connectivity)

**2.3 Java Class Loader**

* Loads compiled Java classes (.class files) into memory for execution.
* Handles dynamic loading, ensuring that required classes are loaded when needed.

**2.4 Memory Management (Heap, Stack, Garbage Collection)**

* JRE manages memory through automatic garbage collection.
* The memory is divided into:
  + Heap Memory (Stores objects created during runtime)
  + Stack Memory (Stores method call details and local variables)

**3. Working of JRE (Step-by-Step)**

**Step 1: Class Loading**

* JRE loads the compiled Java bytecode (.class files) into memory using the Class Loader.
* It ensures that only required classes are loaded, optimizing memory usage.

**Step 2: Bytecode Verification**

* The Bytecode Verifier checks the loaded classes for security and correctness.
* Ensures no unauthorized memory access or invalid operations occur.

**Step 3: Memory Allocation**

* JRE allocates memory dynamically for objects and variables in Heap and Stack Memory.
* It ensures efficient resource management to prevent memory leaks.

**Step 4: Execution of Bytecode**

* The JVM Execution Engine reads the bytecode and translates it into machine code using:
  + Interpreter (Executes bytecode line by line but is slower)
  + JIT (Just-In-Time) Compiler (Optimizes execution by converting bytecode into native code for faster performance)

**Step 5: Garbage Collection**

* JRE automatically removes unused objects from Heap Memory using Garbage Collection (GC).
* This prevents memory overflow and optimizes application performance.

**Step 6: Generating Output**

* After execution, the final output is displayed on the console or GUI.
* JRE ensures smooth execution by handling exceptions, memory allocation, and resource management.

**4. Summary of JRE Working Process**

1. Loads Java classes into memory using the Class Loader.
2. Verifies bytecode for security and correctness.
3. Allocates memory for program execution (Heap & Stack).
4. Executes bytecode using the JVM Execution Engine.
5. Performs garbage collection to free memory.
6. Generates output and handles runtime exceptions.

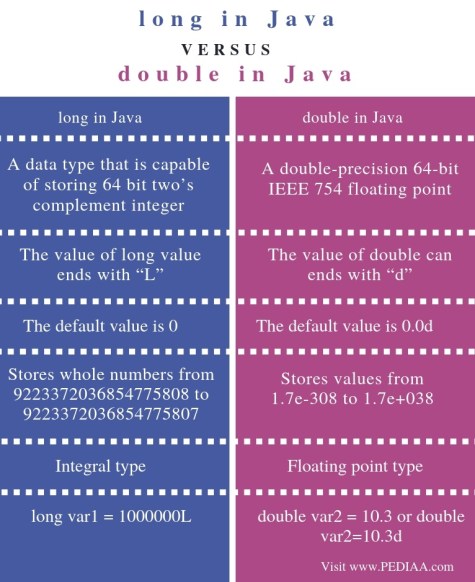
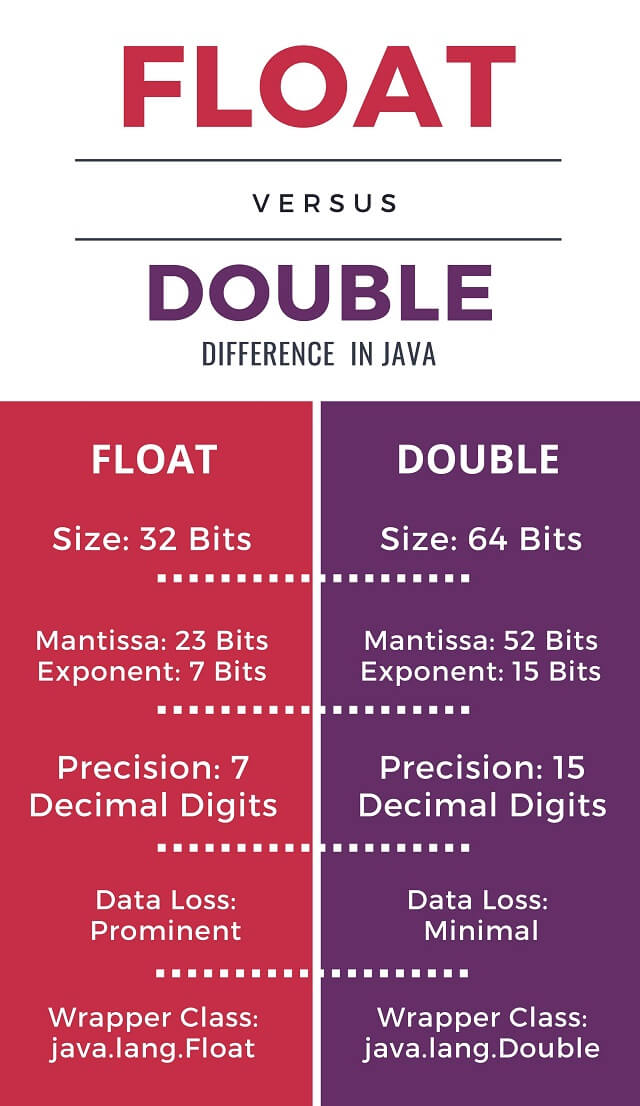
**5. Key Differences: JDK vs JRE vs JVM**

| Feature | JDK (Java Development Kit) | JRE (Java Runtime Environment) | JVM (Java Virtual Machine) |
| --- | --- | --- | --- |
| Includes Compiler (javac) | ✅ Yes | ❌ No | ❌ No |
| Executes Java Code | ✅ Yes | ✅ Yes | ✅ Yes |
| Contains Development Tools | ✅ Yes | ❌ No | ❌ No |
| Provides JVM | ✅ Yes | ✅ Yes | ✅ Yes |
| Handles Memory & Garbage Collection | ✅ Yes | ✅ Yes | ✅ Yes |

6. Conclusion

The JRE is crucial for running Java applications as it provides the required libraries, memory management, and execution environment. It ensures that Java programs run efficiently without requiring additional compilation tools.

1. **Difference between Float and Double.**

1. **Explain Tokens in Detail.**

**Ans : Tokens in Java (Detailed Explanation)**

**Introduction to Java Tokens**

Tokens are the smallest units in a Java program that are meaningful to the compiler. Java has five main types of tokens:

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

1. Keywords

* Keywords are predefined words reserved by Java for specific purposes.
* They cannot be used as variable names, class names, or method names.
* Java has 53 keywords in Java SE 17, including:
  + Data Type Keywords: int, float, double, char, boolean, byte, short, long
  + Control Flow Keywords: if, else, switch, case, default, for, while, do, break, continue, return
  + Class & Object Keywords: class, interface, extends, implements, abstract, final, static
  + Exception Handling Keywords: try, catch, finally, throw, throws
  + Access Modifiers: public, private, protected
  + Other: new, this, super, import, package, synchronized, volatile, assert, instanceof

✅ Allowed:

* Keywords must be in lowercase.

🚫 Not Allowed:

* Using keywords as variable names (int class = 5; is invalid).

2. Identifiers

* Identifiers are names used for variables, methods, classes, interfaces, etc.
* They must follow certain naming rules.

✅ Allowed:

* Can contain letters (A-Z, a-z), digits (0-9), underscore (\_) and dollar sign ($).
* Must begin with a letter, underscore, or dollar sign.

🚫 Not Allowed:

* Cannot use Java keywords (int static = 10; is invalid).
* Cannot start with a digit (int 1name; is invalid).
* No special characters except \_ and $.

3. Literals

* Literals are constant values assigned to variables.
* Types of literals:
  + Integer Literals: Numbers like 10, 255, 0b1010 (binary), 0x1A (hexadecimal).
  + Floating-Point Literals: Decimal numbers like 3.14, 2.0e3.
  + Character Literals: Single characters enclosed in single quotes ('A', '5').
  + String Literals: Sequence of characters in double quotes ("Hello").
  + Boolean Literals: true, false.
  + Null Literal: null, used to represent a reference with no value.

✅ Allowed:

* Underscores in numeric literals (1\_000\_000 is valid).

🚫 Not Allowed:

* Using characters in integer literals (int num = 12A; is invalid).

4. Operators

* Operators perform mathematical, logical, and relational operations.
* Types of operators:
  + Arithmetic Operators: +, -, \*, /, %
  + Relational Operators: ==, !=, >, <, >=, <=
  + Logical Operators: &&, ||, !
  + Bitwise Operators: &, |, ^, <<, >>
  + Assignment Operators: =, +=, -=, \*=, /=, %=
  + Ternary Operator: condition ? value1 : value2
  + Increment/Decrement Operators: ++, --

✅ Allowed:

* Using operators with compatible data types.

🚫 Not Allowed:

* Dividing by zero (5 / 0) results in an ArithmeticException.

5. Separators (Delimiters)

* Java uses special symbols to separate statements and define code blocks.
* Examples:
  + Parentheses () → Used in method calls and control structures.
  + Curly Braces {} → Define code blocks.
  + Square Brackets [] → Used in arrays.
  + Semicolon ; → Marks the end of a statement.
  + Comma , → Used to separate variables or parameters.
  + Dot . → Used to access class members.

✅ Allowed:

* Using ; at the end of every Java statement.

🚫 Not Allowed:

* Missing closing braces (}) will cause compilation errors.

Data Types in Java (Detailed Explanation)

1. Primitive Data Types (8 Types)

| Data Type | Size | Default Value | Allowed Values | Example Values |
| --- | --- | --- | --- | --- |
| byte | 1 byte | 0 | Integer values (-128 to 127) | -100, 50 |
| short | 2 bytes | 0 | Integer values (-32,768 to 32,767) | 3000, -25000 |
| int | 4 bytes | 0 | Integer values (-2^31 to 2^31-1) | 100000, -500 |
| long | 8 bytes | 0L | Large integer values (-2^63 to 2^63-1) | 9999999999L |
| float | 4 bytes | 0.0f | Decimal values (7 decimal digits precision) | 3.14f, -0.1f |
| double | 8 bytes | 0.0d | Decimal values (15 decimal digits precision) | 3.1415926535 |
| char | 2 bytes | \u0000 | Single character | 'A', '#' |
| boolean | 1 bit | false | true or false | true, false |

✅ Allowed:

* Using L for long, f for float, and d for double literals.
* char stores single character enclosed in single quotes.

🚫 Not Allowed:

* Assigning decimal values to integer types (int num = 3.5; is invalid).
* Assigning multiple characters to char (char c = 'AB'; is invalid).

2. Non-Primitive (Reference) Data Types

* These store memory addresses instead of actual values.
* Examples:
  + Strings (String)
  + Arrays (int[], double[])
  + Classes and Objects (Student, Employee)
  + Interfaces and Abstract Classes

✅ Allowed:

* Assigning null to reference variables (String str = null;).

🚫 Not Allowed:

* Accessing null objects (str.length(); on a null String causes NullPointerException).

Summary of Data Type Rules

✅ Allowed  
✔ Using correct literals (long num = 100L;).  
✔ Using boolean only for true/false.  
✔ Using float with f (float pi = 3.14f;).

🚫 Not Allowed  
❌ Mixing incompatible types (boolean b = 1; is invalid).  
❌ Storing decimal values in int.  
❌ Using null with primitive types.

**Day 3:**

**1. ArithmeticOperators.java**

1. **Topic:** Arithmetic Operators in Java
2. **Description:** Arithmetic operators are used to perform mathematical operations such as addition, subtraction, multiplication, and division.
3. **Important Information:**
   1. There are five basic arithmetic operators in Java: +, -, \*, /, and %.
   2. The + operator is used for addition, - for subtraction, \* for multiplication, / for division, and % for modulus (remainder).
   3. The order of operations is followed using the PEMDAS rule: Parentheses, Exponents, Multiplication and Division, and Addition and Subtraction.
   4. Example: int a = 10; int b = 3; int result = a + b; // result = 13

**2. CmdArgs.java and CmdArgs1.java**

1. **Topic:** Command-Line Arguments in Java
2. **Description:** Command-line arguments are used to pass data to a Java program from the command line.
3. **Important Information:**
   1. Command-line arguments are passed to the main method as an array of strings.
   2. The args array contains the command-line arguments, where args[0] is the first argument, args[1] is the second argument, and so on.
   3. Example: public static void main(String[] args) { System.out.println("Hello, " + args[0]); }
   4. To run a Java program with command-line arguments, use the java command followed by the class name and the arguments: java MyClass arg1 arg2

**3. ForDemo.java and ForDemo1.java**

1. **Topic:** For Loops in Java
2. **Description:** For loops are used to repeat a block of code for a specified number of times.
3. **Important Information:**
   1. The basic syntax of a for loop is for (initialization; condition; increment) { code }.
   2. The initialization statement is executed once before the loop starts.
   3. The condition statement is evaluated at the beginning of each iteration. If it is true, the code inside the loop is executed.
   4. The increment statement is executed at the end of each iteration.
   5. Example: for (int i = 0; i < 5; i++) { System.out.println(i); }

**4. InputDemo.java**

1. **Topic:** Input/Output in Java
2. **Description:** Input/output operations are used to read data from the user and display output to the user.
3. **Important Information:**
   1. The Scanner class is used to read input from the user.
   2. The System.out.println() method is used to display output to the user.
   3. Example: Scanner scanner = new Scanner(System.in); System.out.print("Enter your name: "); String name = scanner.nextLine(); System.out.println("Hello, " + name);

**5. LogicalOperators.java**

1. **Topic:** Logical Operators in Java
2. **Description:** Logical operators are used to combine conditional statements.
3. **Important Information:**
   1. There are three logical operators in Java: && (and), || (or), and ! (not).
   2. The && operator returns true if both conditions are true.
   3. The || operator returns true if either condition is true.
   4. The ! operator returns true if the condition is false.
   5. Example: int a = 10; int b = 5; if (a > 5 && b < 10) { System.out.println("Both conditions are true"); }

**6. OperatorDemo.java and OperatorDemo5.java**

1. **Topic:** Operators in Java
2. **Description:** Operators are used to perform various operations such as arithmetic, comparison, logical, and assignment.
3. **Important Information:**
   1. There are several types of operators in Java, including arithmetic, comparison, logical, and assignment operators.
   2. Each operator has a specific syntax and usage.
   3. Example: int a = 10; int b = 5; int result = a + b; // result = 15

**7. RelationalOperators.java**

1. **Topic:** Relational Operators in Java
2. **Description:** Relational operators are used to compare two values.
3. **Important Information:**
   1. There are six relational operators in Java: == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to).
   2. The == operator returns true if both values are equal.
   3. The != operator returns true if both values are not equal.
   4. Example: int a = 10; int b = 5; if (a > b) { System.out.println("a is greater than b"); }

**8. SwitchDemo.java and SwitchDemo1.java**

1. **Topic:** Switch Statements in Java
2. **Description:** Switch statements are used to execute different blocks of code based on the value of a variable.
3. **Important Information:**
   1. The basic syntax of a switch statement is switch (variable) { case value1: code; break; case value2: code; break; ... default: code; }.
   2. The switch statement evaluates the value of the variable and executes the corresponding block of code.
   3. The break statement is used to exit the switch statement.
   4. Example: int day = 2; switch (day) { case 1: System.out.println("Monday"); break; case 2: System.out.println("Tuesday"); break; ... default: System.out.println("Invalid day"); }

**9. TernaryDemo.java**

1. **Topic:** Ternary Operator in Java
2. **Description:** The ternary operator is a shorthand way of writing an if-else statement.
3. **Important Information:**
   1. The basic syntax of the ternary operator is condition ? true\_value : false\_value.
   2. The condition is evaluated, and if it is true, the true\_value is returned. Otherwise, the false\_value is returned.
   3. Example: int a = 10; int b = 5; int result = (a > b) ? a : b; // result = 10

2. **Operators In Java.**

**Ans: Operators in Java (Detailed Explanation with Examples)**

1. Introduction to Operators in Java

Operators in Java are symbols that perform operations on variables and values. Java supports various types of operators categorized based on their functionality.

Types of Operators in Java:

1. Arithmetic Operators
2. Relational (Comparison) Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Ternary Operator
7. Increment and Decrement Operators
8. Instanceof Operator

2. Arithmetic Operators

* Used for performing mathematical calculations.

| Operator | Description | Example (a = 10, b = 3) | Output |
| --- | --- | --- | --- |
| + | Addition | a + b | 13 |
| - | Subtraction | a - b | 7 |
| \* | Multiplication | a \* b | 30 |
| / | Division (Quotient) | a / b | 3 |
| % | Modulus (Remainder) | a % b | 1 |

✅ Allowed:  
✔ Performing arithmetic on numeric data types (int, double, etc.).

🚫 Not Allowed:  
❌ Division by zero (a / 0) causes an ArithmeticException.

3. Relational (Comparison) Operators

* Used to compare values and return true or false.

| Operator | Description | Example (a = 10, b = 5) | Output |
| --- | --- | --- | --- |
| == | Equal to | a == b | false |
| != | Not equal to | a != b | true |
| > | Greater than | a > b | true |
| < | Less than | a < b | false |
| >= | Greater than or equal to | a >= b | true |
| <= | Less than or equal to | a <= b | false |

✅ Allowed:  
✔ Used with int, float, char, etc.  
✔ Works with both positive and negative numbers.

🚫 Not Allowed:  
❌ Cannot compare incompatible types (int == String is invalid).

4. Logical Operators

* Used to perform logical operations (AND, OR, NOT) with boolean values.

| Operator | Description | Example (x = true, y = false) | Output |
| --- | --- | --- | --- |
| && | Logical AND | x && y | false |
| ` |  | ` | Logical OR |
| ! | Logical NOT | !x | false |

✅ Allowed:  
✔ Used with boolean values.

🚫 Not Allowed:  
❌ Cannot use logical operators on non-boolean values (5 && 3 is invalid).

5. Bitwise Operators

* Used to perform bit-level operations on integer types.

| Operator | Description | Example (a = 5 (0101), b = 3 (0011)) | Output |
| --- | --- | --- | --- |
| & | Bitwise AND | a & b | 1 (0001) |
| ` | ` | Bitwise OR | `a |
| ^ | Bitwise XOR | a ^ b | 6 (0110) |
| ~ | Bitwise Complement | ~a | -6 |
| << | Left Shift | a << 1 | 10 |
| >> | Right Shift | a >> 1 | 2 |

✅ Allowed:  
✔ Used with integer data types (byte, short, int, long).

🚫 Not Allowed:  
❌ Cannot use bitwise operators on boolean, float, or double types.

6. Assignment Operators

* Used to assign values to variables.

| Operator | Description | Example (a = 5) | Equivalent To |
| --- | --- | --- | --- |
| = | Assign | a = 5 | - |
| += | Add and Assign | a += 3 | a = a + 3 |
| -= | Subtract and Assign | a -= 2 | a = a - 2 |
| \*= | Multiply and Assign | a \*= 4 | a = a \* 4 |
| /= | Divide and Assign | a /= 2 | a = a / 2 |
| %= | Modulus and Assign | a %= 3 | a = a % 3 |

✅ Allowed:  
✔ Works with all numeric data types (int, double, float, etc.).

🚫 Not Allowed:  
❌ Cannot assign values to final variables (final int x = 5; x = 10; is invalid).

7. Ternary Operator (?:)

* A shorthand for if-else.

Syntax:

java

CopyEdit

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

✅ Example:  
int min = (a < b) ? a : b; // Assigns the smaller value

✅ Allowed:  
✔ Used for conditional value assignments.

🚫 Not Allowed:  
❌ Cannot be used for complex logic (better to use if-else for readability).

8. Increment and Decrement Operators

* Used to increase or decrease a variable’s value by 1.

| Operator | Type | Example (a = 5) | Output |
| --- | --- | --- | --- |
| ++ | Pre-Increment | ++a | 6 |
| ++ | Post-Increment | a++ | 5 (then 6) |
| -- | Pre-Decrement | --a | 4 |
| -- | Post-Decrement | a-- | 5 (then 4) |

✅ Allowed:  
✔ Works with integer and floating-point types.

🚫 Not Allowed:  
❌ Cannot apply to boolean (boolean x = true; x++; is invalid).

9. instanceof Operator

* Checks whether an object belongs to a specific class or subclass.

✅ Example:

java

CopyEdit

if (obj instanceof String) {

System.out.println("It is a String");

}

✅ Allowed:  
✔ Works with objects and inheritance.

🚫 Not Allowed:  
❌ Cannot be used with primitive data types (int instanceof Integer is invalid).

10. Summary of Java Operators

| Type | Operators |
| --- | --- |
| Arithmetic | +, -, \*, /, % |
| Relational | ==, !=, >, <, >=, <= |
| Logical | &&, ` |
| Bitwise | &, ` |
| Assignment | =, +=, -=, \*=, /=, %= |
| Ternary | condition ? true\_value : false\_value |
| Increment/Decrement | ++, -- |
| instanceof | Checks object type |

**Day : 4**

**OOPJ Module - Day 4 Lecture Notes**

**1. Object-Oriented Programming (OOP) Concepts**

**Key Features of OOP:**

1. **Encapsulation:** Wrapping data and methods into a single unit (class).
2. **Abstraction:** Hiding implementation details and exposing only essential features.
3. **Inheritance:** Deriving new classes from existing ones, promoting code reusability.
4. **Polymorphism:** Ability to take multiple forms, allowing method overloading and overriding.

**2. Static, Instance, and Local Variables**

* **Instance Variables:** Variables declared inside a class but outside any method. These are specific to objects.
* **Static Variables:** Variables shared across all instances of a class.
* **Local Variables:** Variables declared inside a method, only accessible within that method.

**Example:**

class Employee {

private int empId; // Instance variable

private String empName; // Instance variable

private static int totalEmployees; // Static variable

void set(int id, String name) {

empId = id;

empName = name;

}

void show() {

System.out.println(empId + " " + empName + " " + totalEmployees);

}

}

**3. Arrays in Java**

**Array Declaration and Initialization**

* **1D Arrays:**

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

* **2D Arrays:**

int arr[][] = new int[3][3];

* **Jagged Arrays:** (arrays with different row sizes)

int arr[][] = new int[3][];

arr[0] = new int[]{1, 2};

arr[1] = new int[]{3, 4, 5};

arr[2] = new int[]{6};

**Example:**

class ArrayDemo {

public static void main(String args[]) {

int arr[][] = {{1, 2, 3}, {11, 12, 13}, {21, 22, 23}};

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

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

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

}

System.out.println();

}

}

}

**Output:**

1 2 3

11 12 13

21 22 23

**4. For-Each Loop**

* Used for iterating over arrays and collections.

for(int a : arr) {

System.out.println(a);

}

**5. Scanner for User Input**

* **Reading input using Scanner**:

import java.util.Scanner;

class ScannerExample {

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

int arr[] = new int[5];

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

System.out.println("Enter element:");

arr[i] = sc.nextInt();

}

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

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

}

}

}

**6. Static Keyword**

* **Static Variables:** Shared across all instances of a class.
* **Static Methods:** Belong to the class and can be called without creating an object.
* **Static Blocks:** Execute when the class is loaded.

**Example:**

class StaticDemo {

static int x = 10;

static void show() {

System.out.println("Static Method: " + x);

}

public static void main(String args[]) {

show();

}

}

**Output:**

Static Method: 10

**7. Understanding Static vs Instance Methods**

* **Instance methods** require an object to be called.
* **Static methods** can be called using the class name.

**Example:**

class Employee {

static void add(int a, int b) {

System.out.println(a + b);

}

}

class EmployeeDemo {

public static void main(String args[]) {

Employee.add(5, 7);

}

}

**Output:**

12

**8. Static vs Non-Static Variable Access**

* **Static variables can be accessed without creating an object.**
* **Instance variables need an object to be accessed.**
* **Non-static variables cannot be accessed in a static method.**

**Incorrect Code:**

class Demo {

int a = 10; // Instance variable

static void show() {

System.out.println(a); // Error: Cannot access non-static variable in a static method.

}

}

**Fix:**

class Demo {

int a = 10;

void show() {

System.out.println(a); // Works fine

}

}

**9. Static Variable Behavior in Multiple Objects**

class Test {

static int counter = 0;

void increment() {

counter++;

}

}

class Main {

public static void main(String[] args) {

Test obj1 = new Test();

Test obj2 = new Test();

obj1.increment();

obj2.increment();

System.out.println("Counter: " + Test.counter);

}

}

**Output:**

Counter: 2

**10. Summary of Key Takeaways**

1. **OOP Concepts:** Encapsulation, Abstraction, Inheritance, Polymorphism.
2. **Variable Types:** Static, Instance, Local.
3. **Array Handling:** 1D, 2D, Jagged arrays, and iteration using for-each.
4. **User Input:** Using Scanner for array input.
5. **Static vs Instance:** Static members belong to the class, instance members belong to objects.
6. **Static Blocks and Methods:** Static members can be accessed without an instance.

These concepts and examples will help in understanding Java’s object-oriented approach and handling arrays effectively.

**Day 5 :**

* 1. **What is Array Introduction.**

**Ans :**

* Arrays in Java are a fundamental data structure used to store collections of elements, **typically of the same type (Homogeneus).** They are objects in Java, which means they are **created on the heap,** and can **hold primitives (like int, char**) or objects (like String, Integer).arrays in Java are objects, which makes them work differently from [arrays in C](https://www.geeksforgeeks.org/c-arrays/)/[C++](https://www.geeksforgeeks.org/cpp-arrays/).
* For **primitive arrays**, **elements**are stored in a contiguous memory location. For **non-primitive arrays**, **references**are stored at contiguous locations, but the actual objects may be at different locations in memory.
* **an array is an object containing a fixed number of values of the same type. The elements of an array are indexed, which means we can access them with numbers**
* **There are primitive type arrays and object type arrays. This means we can use arrays of *int, float, boolean, …* But also arrays of *String, Object* and custom types as well.**
* **Declaration** : int[] anArray; , int anOtherArray[];
* **Initialization :** int[] anArray = new int[10];
* **possibility to set values to the array directly when creating it:**

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

* we initialized a five element array containing numbers 1 to 5. When using this method we don’t need to specify the length of the array, it’s the number of elements then declared between the braces.
* The length of array is fixed and cannot be changed.
* Java provides the built-in property *length* that can be used to determine the length of an array. This property is available for all array types and returns the number of elements in the array:

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

* **Accessing Element** : anArray[0] = 10;

System.out.println(anArray[0]);

* **Create an Array**

To create an array, you need to allocate memory for it using the [new keyword](https://www.geeksforgeeks.org/new-operator-java/):

*// Creating an array of 5 integers  
int[] numbers = new int[5];*

* In Java, all arrays are [dynamically allocated](https://www.geeksforgeeks.org/what-is-dynamic-memory-allocation/).
* Arrays may be stored in contiguous memory [consecutive memory locations].
* Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++, where we find length using size of.
* A Java array variable can also be declared like other variables with [] after the data type.
* The variables in the array are ordered, and each has an index beginning with 0.
* Java array can also be used as a static field, a local variable, or a method parameter.
* An array can contain primitives (int, char, etc.) and object (or non-primitive) references of a class, depending on the definition of the array. In the case of primitive data types, the actual values might be stored in contiguous memory locations (JVM does not guarantee this behavior). In the case of class objects,[the actual objects are stored in a heap segment](https://www.geeksforgeeks.org/g-fact-46/).

**What happens if we try to access elements outside the array size?**

* JVM throws **ArrayIndexOutOfBoundsException** to indicate that the array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of an array.

**Cloning Arrays in Java**

**1. Cloning of Single-Dimensional Array**

When you clone a single-dimensional array, such as Object[], a **shallow copy** is performed. This means that the new array contains references to the original array’s elements rather than copies of the objects themselves. A deep copy occurs only with arrays containing primitive data types, where the actual values are copied.

* **Passing Argument In java.**

**To pass an array as an argument to a method, you just have to pass the name of the array without square brackets.**

**Cloning :** This creates a "shallow" copy, which means that copies of objects in the list are not created, instead the list has references to the same objects that are in the original list.

* **Shallow Copy :** The variables A and B refer to different areas of memory, when B is assigned to A the two variables refer to the same area of memory. Later modifications to the contents of either are instantly reflected in the contents of other, as they share contents.
* Deep Copy : The variables A and B refer to different areas of memory, when B is assigned to A the values in the memory area which A points to are copied into the memory area to which B points. Later modifications to the contents of either remain unique to A or B; the contents are not shared.
* **Why Shallow Copy is fater than Deep Copy :**

In programming, deep copy is equivalent to a physical copy of something. It is an actual copy of the original object. In most programming tools, you can play around with it, modify it without affecting the original object. However, on the other hand, a shallow copy is a reference to the original object. If you change it, it will affect the original object as well. In short, since the deep copy is the actual copy of the original object, it is heavier that the shallow copy which just points to the original object.

**Eamples :** Shallow copy: You can have a picture(s) of your new furniture, and get an idea of what it really looks like. You can easily carry around the picture.

Deep copy: You can go to the furniture shop, and look at the real

* **Stack Memory in Java**
* Stack Memory in Java is used for static memory allocation and the execution of a thread. It contains primitive values that are specific to a method and references to objects referred from the method that are in a heap.
* Access to this memory is in Last-In-First-Out (LIFO) order. Whenever we call a new method, a new block is created on top of the stack which contains values specific to that method, like primitive variables and references to objects.
* When the method finishes execution, its corresponding stack frame is flushed, the flow goes back to the calling method, and space becomes available for the next method.
* **Heap Space in Java**
* **Heap space is used for the dynamic memory allocation of Java objects and JRE classes at runtime**. New objects are always created in heap space, and the references to these objects are stored in stack memory.
* These objects have global access and we can access them from anywhere in the application.

**Constructors**

**Day 6:**

* In Java, a Constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

**Definition**:  
A constructor in Java is a special method that is used to initialize objects when they are created. It has the same name as the class and does not have a return type, not even void. Constructors are called when an object is created using the new keyword.

**Types of Constructors:**

* No-Argument Constructor (Default Constructor):  
  A no-argument constructor is a constructor that does not take any parameters. It is also known as a default constructor. If a class does not have any constructor defined, the compiler provides a default no-argument constructor.
* **Parameterized Constructor:**A parameterized constructor is a constructor that takes one or more parameters. It is used to initialize objects with specific values.
* **Copy Constructor:**  
  A copy constructor is a constructor that creates a copy of an existing object. It is used to create a new object that is a copy of an existing object.
* **Constructor Overloading:**  
  Constructor overloading is a technique where multiple constructors are defined with different parameter lists. This allows objects to be created with different sets of parameters.
* **‘This’ Keyword in java :**
* In Java, ‘this’ is a reference variable that refers to the current object, or can be said “this” in Java is a keyword that refers to the current object instance. It can be used to call current class methods and fields, to pass an instance of the current class as a parameter, and to differentiate between the local and instance variables. Using “this” reference can improve code readability and reduce naming conflicts.
* **Java this reference Example**

In Java, this is a reference variable that refers to the current object on which the method or constructor is being invoked. It can be used to access instance variables and methods of the current object.

**Methods to use ‘this’ in Java**

Following are the ways to use the ‘this’ keyword in Java mentioned below:

* Using the ‘this’ keyword to refer to current class instance variables.
* Using this() to invoke the current class constructor
* Using ‘this’ keyword to return the current class instance
* Using ‘this’ keyword as the method parameter
* Using ‘this’ keyword to invoke the current class method
* Using ‘this’ keyword as an argument in the constructor call
* **Constructor Chaninng :**
* Constructor chaining is a programming technique that allows one constructor to call another constructor. It's used to simplify object initialization and avoid code duplication.
* **Why use constructor chaining?**

**Reuses initialization logic**: Avoids duplicating common setup tasks across multiple constructors

**Simplifies object initialization**: Makes it easier to initialize objects in a controlled and consistent manner

**Makes code easier to read**: Centralizes initialization processes and promotes code reuse

**Avoids code duplication**: Helps to maintain clean and concise code

* **How does constructor chaining work?**

In Java, constructor chaining can occur within the same class or in its superclass

It offers similar benefits to method overloading

It enables you to define multiple constructors and call them with a shared initialization process.

* **Pass By Value :** It means the value of a variable is passed to a method and when a method is called, a new copy of the variable is created in the method’s scope. Any changes made to this variable within the method don’t affect the original variable.
* **Pass-by-reference**: When a method is called, a reference (memory address) to the original variable is passed to the method. Any changes made to the variable within the method affect the original variable as well.
* Primitive data types (int, float, char, etc.) are passed by value.
* Object references are also passed by value. However, because the value being passed is the reference (memory address) of the object, it might feel like it’s being passed by reference.
* **Reassignment of Reference Variables in Java (Detailed Explanation)**
* In Java, **reference variables** store the memory address of objects rather than the actual data itself. When you reassign a reference variable, you are changing the object it points to, not modifying the object itself.
* **How Reference Variables Work in Java**
* **Reference variables store addresses, not values**

When you create an object and assign it to a reference variable, the variable does not hold the actual object but rather the memory address where the object is stored.

* **Reassignment changes the reference, not the object**

If you assign a new object to the reference variable, the variable will now point to the new object, and the old object will be eligible for garbage collection (if no other reference exists).

* **Reassignment does not copy objects**

If two reference variables point to the same object, modifying the object through one reference will affect the other reference.

* **Key Points on Reassigning Reference Variables**

A reference variable can be reassigned to another object of the same type.

When reassigned, the previous object remains in memory only if another reference exists; otherwise, it becomes garbage.

* Primitive type variables in Java do not behave this way because they store actual values, not references.
* If a reference variable is marked as final, it cannot be reassigned to a new object, but the internal state of the object it points to can still be modified.
* **Example Scenario (Without Code)**
* You create an object A and assign it to a reference variable ref1.
* You then assign ref1 to another reference variable ref2, making both refer to the same object.
* If you modify the object using ref2, changes are reflected in ref1 since they point to the same memory location.
* Later, if you reassign ref1 to a new object B, ref2 still refers to A, but ref1 now points to B.
* **Copy Constructor :**
* A copy constructor in Java is a special type of constructor that creates a new object by copying the values of an existing object. It's useful for creating deep copies of objects.
* **How it works**
* **The copy constructor takes an** existing object of the same class as a parameter.
* It copies the values of all member variables, including primitive types, object references, and arrays.
* It creates a separate instance with the same data as the original object.

**Day 7 :**

* **Abstraction :** Abstraction in programming is a methodology that enables developers to focus on the essential qualities of an object rather than its specific details. It is about creating a simple model that represents more complex underlying code and data, simplifying the way interactions with the software are managed. By highlighting what an object does instead of how it achieves what it does, abstraction helps in reducing complexity and enhances the manageability of the software development process.
* **The Importance of Abstraction in Java**

Abstraction is critical in Java for several reasons:

* **Simplification:** It helps in simplifying complex systems by modeling classes based on what they do rather than how they do it. This approach makes it easier for programmers to manage and manipulate the system.
* **Reusability:**By focusing on the interface rather than the implementation, abstraction allows for the creation of reusable code. Different programmers can implement the same interface in various ways to suit their needs.
* **Scalability:** Abstract models can easily be expanded or modified to incorporate new features or behaviors without altering the underlying system’s complexity.
* **Maintenance:** Software maintenance becomes more manageable with abstraction, as changes to the abstracted code have minimal impact on the entire system. This isolation of changes helps in reducing the likelihood of bugs when modifying the code.
* **GPT :Abstraction.**

**What is Abstraction?**

Abstraction is an **Object-Oriented Programming (OOP) principle** that focuses on **hiding implementation details** and exposing only the necessary functionalities to the user. It helps in simplifying complex systems by allowing users to interact with an interface without worrying about the underlying code.

**Why Use Abstraction?**

* **Hides complexity**: Users don’t need to know how the system works internally.
* **Enhances security**: Sensitive implementation details remain hidden.
* **Improves maintainability**: Code changes affect only the internal implementation, not the user interface.
* **Increases flexibility**: Different implementations can be swapped easily without affecting users.

**How to Use Abstraction?**

* **Abstract Classes** (abstract keyword in Java, C++)
* **Interfaces** (fully abstract structures in Java, Python, C#)

By using these, we can create a blueprint where child classes provide concrete implementations for abstract methods.

**Real-Time Example of Abstraction**

**ATM Machine**

* When using an **ATM**, you interact with a simple interface (insert card, enter PIN, withdraw money).
* The **internal operations** (verification, database checks, cash dispensing) are hidden from the user.
* The user only cares about the **functionality**, not how it's implemented.
* **Interface in Java**
* An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
* The interface in Java is *a mechanism to achieve*[*abstraction*](https://www.tpointtech.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.tpointtech.com/inheritance-in-java).
* In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
* Java Interface also represents the IS-A relationship.
* It cannot be instantiated just like the abstract class.
* Since Java 8, we can have default and static methods in an interface.
* Since Java 9, we can have private methods in an interface.
* Why use Java interface?
* There are mainly three reasons to use interface. They are given below.
* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.
* **How to declare an interface?**
* An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.
* **Relationship Between Classes and Interfaces**

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



* **Encapsulation :**Encapsulation is a technique that hides the implementation details of a class from other classes. This means that other classes cannot access or modify the data members of a class directly. Instead, they have to use public methods provided by the class to access or modify its data members.

class Person {  
 private String name;  
 private int age;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Person person = new Person();  
 person.setName("John");  
 person.setAge(30);  
 System.out.println("Name: " + person.getName());  
 System.out.println("Age: " + person.getAge());  
 }  
}

In this example, we have created a Person class with two private data members: name and age. We have also provided two public methods (getName and getAge) to access these data members, and two public methods (setName and setAge) to modify them.

When we create an object of the Person class, we can set its name and age using the setName and setAge methods respectively. We can also get its name and age using the getName and getAge methods respectively.

* **Inheritance**
* Inheritance is a way of creating a new class from an existing class. The new class is called the subclass, and the existing class is called the superclass. The subclass inherits all the properties and methods of the superclass. This means that the subclass can use all the methods and properties of the superclass without having to redefine them.
* The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class)

**Need of Java Inheritance**

* **Code Reusability**: The basic need of an inheritance is to reuse the features. If you have defined some functionality once, by using the inheritance you can easily use them in other classes and packages.
* **Extensibility**: The inheritance helps to extend the functionalities of a class. If you have a base class with some functionalities, you can extend them by using the inheritance in the derived class.
* **Implantation of Method Overriding**: Inheritance is required to achieve one of the concepts of Polymorphism which is Method overriding.
* **Achieving Abstraction**: Another concept of OOPs that is abstraction also needs inheritance.
* Types of Java Inheritance
* In Java, there are mainly three types of inheritances **Single**, **Multilevel**, and **Hierarchical**. Java does not support **Multiple** and **Hybrid** inheritance.



In Java, while technically supporting only single and multiple inheritance through interfaces, we can broadly categorize inheritance into Single Inheritance, Multilevel Inheritance, Hierarchical Inheritance, Multiple Inheritance (through interfaces), and Hybrid Inheritance. Let's explore each of them in detail:

1. Single Inheritance:

* **Concept:** A subclass inherits from only one superclass. It's the most straightforward form of inheritance.
* **Example:** If we have a class Animal and a class Dog that extends Animal, Dog would inherit the properties and methods of Animal.
* **Implementation:**

Java

class Animal {  
 public void makeSound() {  
 System.out.println("Animal sound");  
 }  
 }  
  
 class Dog extends Animal {  
 public void bark() {  
 System.out.println("Woof!");  
 }  
 }

2. Multilevel Inheritance:

* **Concept:** A subclass inherits from another subclass, which itself inherits from a parent class, creating a hierarchy or chain of inheritance.
* **Example:** If Animal is the base class, Dog inherits from Animal, and Labrador inherits from Dog, then Labrador inherits from both Dog and indirectly from Animal.
* **Implementation:**

Java

class Animal {  
 public void makeSound() {  
 System.out.println("Animal sound");  
 }  
 }  
  
 class Dog extends Animal {  
 public void bark() {  
 System.out.println("Woof!");  
 }  
 }  
  
 class Labrador extends Dog {  
 public void fetch() {  
 System.out.println("Fetching the ball!");  
 }  
 }

3. Hierarchical Inheritance:

* **Concept:** Multiple subclasses inherit from a single parent class.
* **Example:** If Animal is the base class, then classes like Dog, Cat, and Bird could all extend Animal, each inheriting the properties and methods of Animal.
* **Implementation:**

Java

class Animal {  
 public void makeSound() {  
 System.out.println("Animal sound");  
 }  
 }  
  
 class Dog extends Animal {  
 public void bark() {  
 System.out.println("Woof!");  
 }  
 }  
  
 class Cat extends Animal {  
 public void meow() {  
 System.out.println("Meow!");  
 }  
 }  
  
 class Bird extends Animal {  
 public void chirp() {  
 System.out.println("Chirp!");  
 }  
 }

4. Multiple Inheritance (through Interfaces):

* **Concept:** In Java, a class can implement multiple interfaces, which is the closest equivalent to multiple inheritance that Java supports. Interfaces define a contract of methods that a class must implement.
* **Example:** A class can implement Flyable and Swimable interfaces and implement methods defined in those interfaces.
* **Implementation:**

Java

interface Flyable {  
 void fly();  
 }  
  
 interface Swimable {  
 void swim();  
 }  
  
 class Bird implements Flyable, Swimable {  
 public void fly() {  
 System.out.println("Flying");  
 }  
 public void swim() {  
 System.out.println("Swimming");  
 }  
 }

5. Hybrid Inheritance:

* **Concept:** A combination of different inheritance types, typically involving a mix of single, multilevel, and multiple inheritance (through interfaces).
* **Example:** A class might inherit from a single class, implement multiple interfaces, and in turn have a subclass which implements another interface.
* **What is Ambiguty in java**

In Java, ambiguity occurs when the compiler cannot definitively determine which method or variable to use in a given context because of multiple, similar choices, leading to errors such as "method X is ambiguous for type Y". This can occur due to method overloading, inheritance, or variable scoping.

Here's a breakdown of common scenarios causing ambiguity in Java:

* **Method Overloading:**

When multiple methods with the same name but different parameters are defined within a class, and the compiler cannot determine which one to call based on the arguments provided in a method call, ambiguity occurs.

* + **Example:** If you have foo(String) and foo(Integer), and you call foo(null), the compiler may not know which one to choose since null can be assigned to both String and Integer.
  + **Solution:** Make sure your method calls are unambiguous, by using an appropriate argument data type.
* **Method Overriding and Inheritance:**

In the case of inheritance, when a derived class inherits methods from multiple superclasses that share the same signature, the compiler may be unable to determine which method should be called by the derived class.

* + **The diamond problem**: occurs with multiple inheritance, where the child class inherits from two or more parent classes that have a method with the same name, so the child class does not know which parent's method to inherit.
  + **Solution:** Java does not support multiple inheritance of classes to avoid the diamond problem.
  + **Example**: If classes A and B both have method doSomething() and class C inherits both classes, and C does not define its own doSomething(), the compiler does not know which doSomething() it should call, and it will throw an error.
* **Variable Scope:**

When two or more variables with the same name exist in different scopes, the compiler could become confused about which variable is being referenced in a particular context.

* + **Solution:** Make sure your variables have clear names, and use appropriate scope to avoid confusion.
* **Interfaces:**

Interfaces in Java are an alternative to multiple inheritance, providing method signatures without implementations, which avoid the ambiguity that multiple inheritance of classes can cause.

* + **Solution:** Implementing multiple interfaces does not lead to ambiguity, as the implementation details must be provided in the derived class.
* **Nested Interface**

We can declare an interface in another interface or class. Such an interface is termed as a nested interface.

**The following are the rules governing a nested interface.**

* A nested interface declared within an interface must be public.
* A nested interface declared within a class can have any access modifier.
* A nested interface is by default static.

**Following is an example of a nested interface.**

**Example**

[**Live Demo**](http://tpcg.io/qjfl8p)

class Animal {

   interface Activity {

      void move();

   }

}

class Dog implements Animal.Activity {

   public void move() {

      System.out.println("Dogs can walk and run");

   }

}

public class Tester {

   public static void main(String args[]) {

      Dog dog = new Dog();

      dog.move();

   }

}

Output

Dogs can walk and run

* **Association & Aggrigation**

**1. Association**

**Definition**

* Association represents a relationship between two or more objects.
* It defines how objects communicate with each other, but **no ownership** exists between them.
* It is a **"has-a"** relationship, meaning one object **has** another, but both can exist independently.

**Types of Association**

1. **One-to-One (1:1)**
   * Example: A **Person** has one **Passport**.
   * Real-world Example: A **CEO** and the company's **official car**.
2. **One-to-Many (1:M) or Many-to-One (M:1)**
   * Example: A **Teacher** teaches multiple **Students**, but each student has only one teacher.
   * Real-world Example: A **Manager** manages multiple **Employees**.
3. **Many-to-Many (M:M)**
   * Example: A **Student** can enroll in multiple **Courses**, and a **Course** can have multiple students.
   * Real-world Example: A **Doctor** treating multiple **Patients**, and a **Patient** consulting multiple doctors.

**Code Example (Java)**

java

CopyEdit

class Person {

String name;

Passport passport; // One-to-One Association

}

class Passport {

String passportNumber;

}

👉 **Here,** Person has a Passport, but both can exist independently.

**2. Aggregation (Weak Association)**

**Definition**

* Aggregation is a **special type of association** where one class contains another class but does **not own it**.
* The contained object (part) can exist **independently** of the container (whole).
* It is a **"has-a"** relationship with a **weak bonding**.
* If the parent object is destroyed, the child object **still exists**.

**Example:**

A **University** has multiple **Departments**, but if the university closes, the departments may still exist.

**Real-world Example:**

* A **Library** has **Books**, but books exist even if the library closes.
* A **Car** has an **Engine**. If the car is sold or scrapped, the engine can still function in another car.

**Code Example (Java)**

java

CopyEdit

class Engine {

String type = "Petrol Engine";

}

class Car {

String model;

Engine engine; // Aggregation (Car has an Engine)

Car(String model, Engine engine) {

this.model = model;

this.engine = engine;

}

}

public class Main {

public static void main(String[] args) {

Engine engine = new Engine();

Car car = new Car("Toyota", engine);

System.out.println(car.model + " has " + car.engine.type);

}

}

**Output:**

nginx

CopyEdit

Toyota has Petrol Engine

👉 **Here,** the Engine exists separately and can be used by another car.

**3. Composition (Strong Association)**

**Definition**

* Composition is a **strong form of Aggregation**, where the contained object **cannot exist independently** of the container object.
* It is a **"has-a"** relationship with **strong ownership**.
* If the parent object is destroyed, the child object **also gets destroyed**.

**Example:**

A **House** has **Rooms**. If the house is destroyed, the rooms cannot exist separately.

**Real-world Example:**

* A **Human Body** has a **Heart**. If the body dies, the heart is useless.
* A **Laptop** has a **Processor**. If the laptop is broken, the processor is no longer useful.

**Code Example (Java)**

java

CopyEdit

class Heart {

String function = "Pumping blood";

}

class Human {

String name;

Heart heart; // Composition (Human has a Heart)

Human(String name) {

this.name = name;

this.heart = new Heart(); // Creating the object within the constructor

}

}

public class Main {

public static void main(String[] args) {

Human person = new Human("John");

System.out.println(person.name + "'s heart is responsible for " + person.heart.function);

}

}

**Output:**

rust

CopyEdit

John's heart is responsible for Pumping blood

👉 **Here,** the Heart object is created inside the Human class. If Human is destroyed, the Heart is also destroyed.

**Key Differences:**

| **Feature** | **Association** | **Aggregation** | **Composition** |
| --- | --- | --- | --- |
| Relationship | General relationship between objects | Whole-part relationship (Weak) | Whole-part relationship (Strong) |
| Ownership | No ownership | Weak ownership | Strong ownership |
| Dependency | Objects can exist independently | Contained object can exist independently | Contained object **cannot** exist independently |
| Example | Teacher and Students | Library and Books | Human and Heart |
| When Parent is Destroyed | Child survives | Child survives | Child is also destroyed |

**Summary**

* **Association**: A general relationship where objects are linked but can exist separately.
* **Aggregation**: A weak relationship where one object contains another, but the contained object can exist independently.
* **Composition**: A strong relationship where one object contains another, and the contained object **cannot** exist without the container.
* **Upcasting Downcasting**
* **Upcasting In Java and How Does It work?**
* It is a process of converting a subclass object to its superclass reference type. In other words, it is a way of treating an object of a subclass as an object of its superclass. This conversion is done implicitly by the Java Virtual Machine (JVM) at runtime.
* When upcasting is performed, the subclass object loses its specific attributes and methods that are not present in its superclass. The resulting object can only access the methods and attributes that are defined in the superclass.
* Upcasting is useful when you want to use a single reference type to refer to objects of different subclasses. This allows you to write more generic code that can handle different types of objects without having to write separate code for each type.
* **Example of Upcasting in Java**
* Let’s consider an example of upcasting in Java. Suppose we have a class hierarchy that looks like this:
* class Animal   
  {   
  public void makeSound()   
  { System.out.println("Animal is making a sound"); }  
  }  
  class Dog extends Animal   
  {  
  public void makeSound()   
  { System.out.println("Dog is barking");   
  }  
  public void fetch()   
  {  
  System.out.println("Dog is fetching");  
  }  
  }   
  Dog myDog = new Dog(); Animal myAnimal = myDog;
* In this code, we create a Dog object called myDog. We then upcast myDog to an Animal reference called myAnimal. Now, myAnimal can only access the methods and attributes that are defined in the Animal class:
* myAnimal.makeSound(); // prints “Dog is barking”
* ***Good to Read:-***[***Easiest Way to Learn What Is Garbage Collection In Java***](https://www.devstringx.com/what-is-garbage-collection-in-java)
* **Downcasting in Java and How Does It Work?**
* Downcasting is the opposite of upcasting. It is a process of converting a superclass reference type to its subclass object. In other words, it is a way of treating an object of a superclass as an object of its subclass. This conversion is done explicitly by the programmer using the cast operator.
* When downcasting is performed, the superclass reference is checked to see if it refers to an object of the subclass. If it does, the reference is converted to the subclass type. If it doesn’t, a ClassCastException is thrown at runtime.
* Downcasting is useful when you want to access the specific methods and attributes of a subclass object that are not present in its superclass. However, it should be used with caution because it can lead to runtime errors if the superclass reference does not refer to an object of the subclass.
* **Example of Downcasting in Java**
* Let’s consider an example of downcasting in Java. Suppose we have the same class hierarchy as before:
* class Animal  
    
  {  
    
  public void makeSound()  
    
  {  
    
  System.out.println("Animal is making a sound");  
    
  }  
    
  }  
    
  class Dog extends Animal  
    
  {  
    
  public void makeSound()  
    
  { System.out.println("Dog is barking");  
    
  }  
    
  public void fetch() {  
    
  System.out.println("Dog is fetching");  
    
  }  
    
  }
* Now, let’s create an Animal object and downcast it to a Dog reference:
* Animal myAnimal = new Dog(); Dog myDog = (Dog) myAnimal;
* In this code, we create an Animal object called myAnimal. We then downcast myAnimal to a Dog reference called myDog. Now, myDog can access the specific methods and attributes that are defined in the Dog class:
* myDog.makeSound(); // prints “Dog is barking” myDog.fetch(); // prints “Dog is fetching”
* **Difference Between Upcasting and Downcasting**
* The main difference between upcasting and downcasting is the direction of the conversion. Upcasting converts a subclass object to its superclass reference, while downcasting converts a superclass reference to its subclass object.
* Upcasting is done implicitly by the JVM at runtime, while downcasting is done explicitly by the programmer using the cast operator.
* It is safe because it only allows access to the methods and attributes that are defined in the superclass. Downcasting, on the other hand, can be dangerous because it can lead to runtime errors if the superclass reference does not refer to an object of the subclass.
* **Difference between is a and has a Relashionship.**

| **Feature** | **IS-A Relationship (Inheritance)** | **HAS-A Relationship (Composition/Aggregation)** |
| --- | --- | --- |
| **Definition** | One class **inherits** another class. | One class **contains** another class as a field (object reference). |
| **Type** | **Generalization-Specialization** (Parent-Child relationship). | **Whole-Part relationship** (Containment). |
| **Implementation** | Implemented using **extends (Java)**, **: (Python, C++)**. | Implemented using **object references (instance variables)** inside a class. |
| **Dependency** | Child class is **dependent** on the Parent class. | The contained object **may** or **may not** depend on the container object. |
| **Tightly Coupled?** | **Yes** (Child is tightly coupled to Parent). | **No (Aggregation)**, **Yes (Composition)**. |
| **Code Reusability** | **High** (inherits methods and attributes). | **Moderate** (reuses an object instead of inheriting). |
| **When to Use?** | When one class is a **specialized version** of another class. | When one class **contains another class** as a part of it. |
| **Example** | Car IS-A Vehicle (Car inherits Vehicle). | Car HAS-A Engine (Car contains Engine object). |
| **Code Example (Java)** | class Car extends Vehicle {} | class Car { Engine engine; } |
| **Real-World Example** | Dog IS-A Animal (A dog is a type of animal). | Library HAS-A Book (A library contains books). |

**Key Takeaways**

* **Use IS-A** (extends) when you need **inheritance** (one class is a type of another).
* **Use HAS-A** (composition or aggregation) when one class **contains** another but doesn't need inheritance.
* **Polymorphism :**

Polymorphism in Java is one of the main aspects of Object-Oriented Programming(OOP). The word polymorphism can be broken down into Poly and morphs, as “Poly” means many and “Morphs” means forms. In simple words, we can say that the ability of a message to be represented in many forms.

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

**Real-Life Examples of Polymorphism**

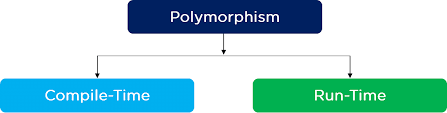
An individual can have different relationships with different people. A woman can be a mother, a daughter, a sister, and a friend, all at the same time, i.e. she performs other behaviors in different situations.

The human body has different organs. Every organ has a different function to perform; the heart is responsible for blood flow, the lungs for breathing, the brain for cognitive activity, and the kidneys for excretion. So we have a standard method function that performs differently depending upon the organ of the body.

**Types of Java Polymorphism**

In Java Polymorphism is mainly divided into two types:

* Compile-time Polymorphism
* Runtime Polymorphism



**Compile-Time Polymorphism in Java**

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading.

**Method Overloading**

Method overloading in Java refers to the ability to define multiple methods in the same class with the same name but with different parameters. Java compiler differentiates these methods based on the number of parameters or the data types of parameters.

Example

public class Calculator {

*// Method to add two integers*

public int add(int *a*, int *b*) {

*return* a + b;

}

*// Method to add three integers*

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

*return* a + b + c;

}

*// Method to add two doubles*

public double add(double *a*, double *b*) {

*return* a + b;

}

*// Method to concatenate two strings*

public String add(String *a*, String *b*) {

*return* a + b;

}

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

Calculator calculator = *new* Calculator();

*// Calling the overloaded methods*

System.out.println("Sum of 5 and 10 is: " + calculator.add(5, 10));

System.out.println("Sum of 5, 10, and 15 is: " + calculator.add(5, 10, 15));

System.out.println("Sum of 3.5 and 2.7 is: " + calculator.add(3.5, 2.7));

System.out.println("Concatenation of 'Hello' and 'World' is: " + calculator.add("Hello", "World"));

}

}

Java

Output

Sum of 5 and 10 is*:* 15

Sum of 5, 10, and 15 is*:* 30

Sum of 3.5 and 2.7 is*:* 6.2

Concatenation of 'Hello' and 'World' is*:* HelloWorld

Java

**Runtime Polymorphism in Java**

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

**Method Overriding**

Method overriding in Java occurs when a subclass provides 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 parameter list as the method in the superclass. Method overriding is used to provide specific behavior for a method in a subclass.

Example

class Animal {

public void makeSound() {

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

}

}

class Dog extends Animal {

@Override

public void makeSound() {

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

}

}

class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("Cat meows");

}

}

public class Main {

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

Animal animal1 = *new* Animal();

Animal animal2 = *new* Dog();

Animal animal3 = *new* Cat();

animal1.makeSound(); *// Output: Animal makes a sound*

animal2.makeSound(); *// Output: Dog barks*

animal3.makeSound(); *// Output: Cat meows*

}

}

Java

Output

Animal makes a sound

Dog barks

Cat meows

Java

**Advantages of Polymorphism in Java**

* **Code Reusability**: Polymorphism allows methods to be reused across different classes, reducing the need for redundant code.
* **Flexibility and Extensibility**: Polymorphism enables the development of flexible and extensible code. New classes can be added without modifying existing code, as long as they adhere to the required interfaces or inherit from existing classes.
* **Simplification of Code**: Polymorphism simplifies code by allowing methods to be defined in superclasses and overridden in subclasses. This promotes a clean and organized class hierarchy.
* **Enhanced Maintainability**: Since polymorphism encourages code reuse and modularization, it enhances code maintainability. Changes made in one part of the codebase may have minimal impact on other parts, making it easier to manage and debug.
* **Dynamic Method Dispatch**: Polymorphism enables dynamic method dispatch, where the appropriate method implementation is determined at runtime based on the actual object type. This promotes flexibility and runtime polymorphism.
* **Support for Interfaces**: Polymorphism facilitates the use of interfaces, allowing objects of different classes to be treated interchangeably based on shared behavior defined by interfaces. This promotes loose coupling and abstraction.
* **Facilitation of Overloading and Overriding**: Polymorphism supports method overloading and overriding, allowing for the creation of multiple methods with the same name but different behaviors. This promotes code clarity and flexibility.
* **Encouragement of Design Patterns**: Polymorphism is central to many design patterns in Java, such as the Strategy pattern, Factory pattern, and Decorator pattern. Leveraging polymorphism in design patterns leads to more maintainable and scalable software architectures.

**Disadvantages of Polymorphism in Java**

* **Runtime Overhead**: Dynamic method dispatch, which is central to polymorphism, can incur runtime overhead due to the additional lookup required to determine the appropriate method implementation. This overhead may be negligible for small programs but can become significant in performance-critical applications.
* **Complexity**: Polymorphism can introduce complexity, especially in large codebases or when used improperly. Understanding which method implementation will be invoked at runtime may require tracing through multiple classes and hierarchies, leading to code that is harder to comprehend and maintain.
* **Hidden Behavior**: Polymorphism can lead to hidden behavior, where the actual method implementation called may not be immediately evident from the code. This can make debugging and troubleshooting more challenging, particularly for developers unfamiliar with the codebase.
* **Performance Impact**: While polymorphism promotes flexibility and code reuse, it can sometimes lead to performance degradation, especially when method invocations involve numerous virtual method calls or when complex inheritance hierarchies are present. This performance impact may necessitate optimization efforts.
* **Potential for Inefficiency**: In certain scenarios, polymorphism may result in inefficient code. For example, method invocations through polymorphic references may incur additional memory overhead compared to direct method calls, as they may involve extra indirection through method tables or vtables.
* **Compiler Optimizations Limitation**: Compiler optimizations for polymorphic code can be limited, particularly in the presence of dynamic class loading, reflection, or complex inheritance structures. This may restrict the extent to which the compiler can optimize method dispatch and may impact runtime performance.
* **Difficulty in Understanding Control Flow**: Polymorphism can make it harder to understand the control flow of a program, especially when multiple classes override the same method. Determining which method implementation will be executed at runtime requires careful analysis of the inheritance hierarchy and object types involved.
* **Potential for Inheritance Issues**: Inheritance hierarchies used in polymorphic code can sometimes lead to design issues such as the fragile base class problem or the diamond problem. These issues can complicate code maintenance and evolution, particularly in large and evolving codebases.

**Difference between Compile-time and Run-time Polymorphism in Java**

|  |  |  |
| --- | --- | --- |
| **Compile Time Polymorphism** | **Run time Polymorphism** |  |
| * In run-time Polymorphism, the call is not resolved by the compiler. | It is also known as Static binding, Early binding, and overloading as well. |  |
| * It is also known as Dynamic binding, Late binding, and overriding as well. | Method overriding is the runtime polymorphism having the same method with the same parameters or signature but associated with compared, different classes. |  |
| * It provides slow execution as compared to early binding because the method that needs to be executed is known at the runtime. | Method overloading is the compile-time polymorphism where more than one methods share the same name with different parameters or signatures and different return types. |  |
| * It is achieved by function overloading and operator overloading. | It is achieved by virtual functions and pointers. |  |
| * It provides fast execution because the method that needs to be executed is known early at the compile time. | It provides slow execution as compare to early binding because the method that needs to be executed is known at the runtime. |  |
| * Compile time polymorphism is less flexible as all things execute at compile time. | Run time polymorphism is more flexible as all things execute at run time. |  |
| Inheritance is not involved. | Inheritance is involved. |  |

* **Difference Between Static and Dynamic Biniding**

| **Sr. No.** | **Key** | **Static Binding** | **Dynamic Binding** |
| --- | --- | --- | --- |
| 1 | Basic | It is resolved at compile time | It is resolved at run time |
| 2 | Resolve mechanism | static binding use type of the class and fields | Dynamic binding uses object to resolve binding |
| 3 | Example | Overloading is an example of static binding | Method overriding is the example of Dynamic binding |
| 4. | Type of Methods | private, final and static methods and variables uses static binding | Virtual methods use dynamic binding |

* **Method Overloading :**

**Introoduction**

Method overloading is a function in Java that allows a category to have a couple of method having the same name, if their parameter lists are one of a kind. It is a shape of collect-time polymorphism where the compiler determines which method to execute primarily based at the technique signature.

**Key Points**

* **Same Method Name**: Multiple strategies in the same magnificence may have the identical call if they have one of a kind parameter lists (exceptional variety of parameters or different varieties of parameters).
* **Return Type**: Method overloading could have the identical or extraordinary go back kinds, however the parameter listing need to vary.
* **Access Modifier**: Overloaded methods may have special get entry to modifiers (e.G., public, included, personal), but they cannot have more restrictive get admission to than the overridden technique.
* **Exception Handling**: Overloaded techniques can claim specific checked exceptions or no exceptions in any respect.

**Example of Method Overloading**

class Calculator {

// Method to add two integers

public int add(int a, int b) {

return a + b;

}

// Method to add three integers

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

return a + b + c;

}

// Method to add two doubles

public double add(double a, double b) {

return a + b;

}

// Method to concatenate two strings

public String add(String a, String b) {

return a + b;

}

public static void main(String[] args) {

Calculator calc = new Calculator();

// Calling overloaded methods

System.out.println("Sum of 5 and 3: " + calc.add(5, 3));

System.out.println("Sum of 5, 3, and 2: " + calc.add(5, 3, 2));

System.out.println("Sum of 2.5 and 3.7: " + calc.add(2.5, 3.7));

System.out.println("Concatenation of 'Hello' and 'World': " + calc.add("Hello", "World"));

}

}

Java

**Output:**

Sum of 5 and 3: 8

Sum of 5, 3, and 2: 10

Sum of 2.5 and 3.7: 6.2

Concatenation of 'Hello' and 'World': HelloWorld

YAML

**Why Do We Need Method Overloading?**

* **Readability**: Method overloading improves code clarity through presenting method names that are intuitive and easy to recollect.
* **Flexibility**: It permits unique techniques to perform similar obligations with differing types or numbers of parameters, enhancing the flexibility of method invocation.
* **Code Reusability**: Overloaded techniques can reuse the good judgment of a way with exclusive parameter configurations, promoting code reusability.
* **Default Values**: It gives a way to outline techniques with default parameter values with out the use of more than one constructors.

**Rules for Method Overloading**

* **Method Signature**: The parameter lists of overloaded techniques must range both within the number of parameters or inside the kind of parameters. Changing simplest the go back kind isn’t always enough.
* **Return Type**: The go back kind may additionally or won’t be distinctive.
* **Access Modifier**: Overloaded strategies may have exceptional get admission to modifiers, however they can not have extra restrictive get right of entry to than the overridden method.
* **Exception Handling**: Overloaded methods can declare extraordinary checked exceptions or no exceptions at all.
* **Method Overriding**

**Introduction**

Method overriding is a fundamental concept in object-oriented programming (OOP) languages, including Java. It allows a subclass to provide its own implementation of a method inherited from its superclass. In this article, we will delve into what method overriding is, why it is essential, provide three examples with code, and highlight the key differences between method overloading and method overriding through a tabular comparison.

**What Is Method Overriding?**

Method overriding occurs when a subclass provides a different implementation of a method that is already defined in its superclass. The overriding method has the same name, return type, and parameters as the method in the superclass. It enables polymorphism, allowing objects of different classes to be treated interchangeably based on their shared superclass.

**Why Is Method Overriding Important?**

 Method overriding serves several purposes and benefits:

* Polymorphism: By overriding methods, you can achieve polymorphism, where different objects can respond differently to the same methods invocation. This flexibility allows you to write more general and reusable code.
* Enhancing Behavior: Method overriding enables you to modify or enhance the behavior of inherited methods in the subclass. This allows you to tailor the behavior of the method to the specific needs of the subclass, adding new functionality or customizing existing behavior.
* Code Extensibility: Method overriding facilitates code extensibility by providing a way to build upon existing functionality. You can inherit and reuse methods from the superclass while adding specific behaviors in the subclass, making your code more modular and maintainable.

**Examples of Method Overriding:**

* Animal Class Hierarchy: In this example, we define an Animal class with a method makeSound() and create subclasses Dog and Cat that override the makeSound() method with their specific implementations.
* Difference Between Method Overriding and Overriding
* **Differences between Method Overloading and Method Overriding:**

|  |  |  |
| --- | --- | --- |
|  | **Method Overloading** | **Method Overriding** |
| **Definition** | Multiple methods with the same name but different parameters | A subclass provides its own implementation of a method inherited from its superclass |
| **Purpose** | To provide variations of a method based on different parameters | To modify or extend the behavior of an inherited method |
| **Signature** | Differs in terms of the number, types, or order of parameters | Remains the same as the overridden method (same name, return type, and parameters) |
| **Relationship** | Same class | Superclass-subclass relationship |
| **Determination** | Compile-time | Run-time (based on the actual type of the object at runtime) |
| **Examples** | Addition of integers and floating-point numbers | Overriding toString() method in Object class, customizing behavior in subclasses like Dog or Cat |
| **Annotation** | Not necessary, as it is resolved during compilation | Optional @Override annotation (provides a compile-time check for correct overriding, ensures method exists in the superclass) |

* **Access Modifiers in java**

**Introduction**

Access modifiers in Java play a crucial role in controlling the accessibility of classes, methods, and variables within a program. Let’s illustrate their importance with a real-world example.

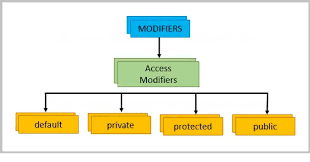
Imagine you’re developing a banking application. In this application, you have different classes representing various components such as accounts, transactions, and customers. Each of these classes has its own set of variables and methods, some of which need to be accessed from outside the class, while others should be kept private for internal use only.

Access modifiers in Java are keywords that are used to control the accessibility or visibility of classes, methods, and variables within a Java program. They determine which other classes can access a particular class, method, or variable.

**Types of Access Modifiers in Java**

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public



**Default Access Modifier**

The default access modifier in Java is used when no access modifier is explicitly specified. It provides package-level access, meaning that classes, methods, and variables with default access are accessible only within the same package.

Here’s a real-world example illustrating the default access modifier:

Let’s say you’re developing a library management system. You have a package **library** containing classes related to managing books, such as **Book**, **Library**, and **Librarian**. These classes have methods and variables that should only be accessed within the **library** package.

package library;

class Book {

String title;

String author;

int pages;

void displayInfo() {

System.out.println("Title: " + title);

System.out.println("Author: " + author);

System.out.println("Pages: " + pages);

}

}

*// Library.java*

package library;

class Library {

*// Some methods and variables*

}

*// Librarian.java*

package library;

class Librarian {

*// Some methods and variables*

}

Java

In this example, the classes **Book**, **Library**, and **Librarian** do not have an access modifier explicitly specified. Therefore, they have default access, which means they are accessible only within the **library** package. If you try to access these classes from outside the **library** package, you’ll get a compilation error.

**Private Access Modifier**

The **private** access modifier in Java restricts the access of a class member (variable or method) only to the class in which it is declared. Here’s a real-world example demonstrating the use of the **private** access modifier:

public class BankAccount {

private String accountNumber; *// Private variable*

private double balance; *// Private variable*

public BankAccount(String *accountNumber*) {

this.accountNumber = accountNumber;

this.balance = 0.0; *// Initialize balance to 0*

}

public void deposit(double *amount*) {

*if* (amount > 0) {

balance += amount;

System.out.println(amount + " deposited into account " + accountNumber);

}

}

public void withdraw(double *amount*) {

*if* (amount > 0 && amount <= balance) {

balance -= amount;

System.out.println(amount + " withdrawn from account " + accountNumber);

} *else* {

System.out.println("Insufficient funds");

}

}

public double getBalance() {

*return* balance;

}

}

*// BankApp.java*

public class BankApp {

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

BankAccount account = *new* BankAccount("1234567890");

*// We cannot directly access the private members of BankAccount*

*// account.accountNumber = "9876543210"; // Compilation error: accountNumber has private access in BankAccount*

*// account.balance = 10000; // Compilation error: balance has private access in BankAccount*

*// But we can interact with BankAccount using its public methods*

account.deposit(500);

account.withdraw(200);

System.out.println("Current Balance: " + account.getBalance());

}

}

Java

In this example:

* The **BankAccount** class has two private variables, **accountNumber** and **balance**. These variables are encapsulated within the class and can only be accessed and modified by methods within the same class.
* The **BankApp** class cannot directly access the private variables **accountNumber** and **balance** of the **BankAccount** class. Attempting to do so would result in a compilation error.
* However, **BankApp** can interact with the **BankAccount** object using its public methods such as **deposit**, **withdraw**, and **getBalance**. These methods internally access and modify the private variables of the **BankAccount** class, maintaining data encapsulation and ensuring controlled access to sensitive data.

**Protected Access Modifier**

The **protected** access modifier in Java allows the member (variable or method) to be accessed within the same package or by subclasses, even if they are in different packages. Here’s a real-world example demonstrating the use of the **protected** access modifier:

Suppose we are building a simple system for a school. We have a superclass **Person** representing a person’s basic information, and subclasses **Student** and **Teacher** representing specific types of people in the school.

package school;

public class Person {

protected String name; *// Protected variable*

protected void introduce() { *// Protected method*

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

}

}

*// Student.java*

package school;

public class Student extends Person {

private int studentId; *// Private variable*

public Student(String *name*, int *studentId*) {

this.name = name; *// Accessing protected member from superclass*

this.studentId = studentId;

}

public void displayInfo() {

introduce(); *// Accessing protected method from superclass*

System.out.println("I am a student with ID " + studentId);

}

}

*// Teacher.java*

package school;

public class Teacher extends Person {

private String subject; *// Private variable*

public Teacher(String *name*, String *subject*) {

this.name = name; *// Accessing protected member from superclass*

this.subject = subject;

}

public void displayInfo() {

introduce(); *// Accessing protected method from superclass*

System.out.println("I am a teacher of " + subject);

}

}

*// SchoolApp.java*

package app;

import school.Student;

import school.Teacher;

public class SchoolApp {

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

Student student = *new* Student("John", 123);

student.displayInfo();

Teacher teacher = *new* Teacher("Jane", "Math");

teacher.displayInfo();

}

}

Java

In this example:

* The **Person** class has a protected variable **name** and a protected method **introduce()**. These members can be accessed by subclasses such as **Student** and **Teacher**.
* The **Student** and **Teacher** classes extend the **Person** class, allowing them to inherit the protected members.
* Both **Student** and **Teacher** classes use the **name** variable and **introduce()** method from the **Person** superclass.
* The **SchoolApp** class creates instances of **Student** and **Teacher** and calls their respective **displayInfo()** methods, which internally use the protected members inherited from the superclass **Person**.

**Public Access Modifier**

The **public** access modifier in Java allows a class, method, or variable to be accessible from any other class, regardless of the package it belongs to. Here’s a real-world example demonstrating the use of the **public** access modifier:

package library;

public class Book {

public String title;

public String author;

public int pages;

public void displayInfo() {

System.out.println("Title: " + title);

System.out.println("Author: " + author);

System.out.println("Pages: " + pages);

}

}

*// Library.java*

package library;

public class Library {

public void addBookToCatalog(Book *book*) {

*// Logic to add the book to the library catalog*

}

}

*// LibraryApp.java*

package app;

import library.Book;

import library.Library;

public class LibraryApp {

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

*// Creating an instance of Book and accessing its public members*

Book book = *new* Book();

book.title = "Java Programming";

book.author = "John Doe";

book.pages = 300;

book.displayInfo();

*// Creating an instance of Library and calling its public method*

Library library = *new* Library();

library.addBookToCatalog(book);

}

}

Java

In this example:

* The **Book** class has its attributes (**title**, **author**, **pages**) and method (**displayInfo()**) declared with the **public** access modifier. This allows them to be accessed from any other class.
* The **Library** class has a method **addBookToCatalog(Book book)** with the **public** access modifier, allowing it to be accessed from any other class. This method accepts a **Book** object as a parameter, demonstrating how public access allows interaction between different classes.
* The **LibraryApp** class in a different package **app** can create instances of **Book** and **Library** and interact with their public members. It demonstrates how classes from different packages can access **public** members of other classes.

**Understanding Java Access Modifiers**



**Day 9**

* **‘Super’ Keyword in java :**

**Introduction**

In Java programming, the super keyword stands as a powerful and versatile element, offering developers a means to navigate and manipulate class hierarchies. The super keyword is used to refer to the superclass or parent class, providing access to its members and allowing for the invocation of its methods and constructors. This feature is fundamental in scenarios where a subclass inherits from a superclass, allowing for the extension and customization of functionality. This exploration delves into the nuances of the super keyword in Java, uncovering its applications, syntax, and the pivotal role it plays in facilitating efficient and organized object-oriented programming.

The ‘super’ keyword allows referencing the parent class or superclass of a subclass in Java. It is often employed to access members (fields or methods) of the superclass that have been overridden in the subclass. You can call the superclass’s method from within the subclass using super.methodName(). Additionally, super() is used to call the constructor of the superclass from the subclass constructor, which is essential for initializing inherited members. In short, if any programmers want to maintain inheritance hierarchies and enable the reuse of code in object-oriented programming, this super keyword is crucial.

**Characteristics of Super Keyword in Java**

In Java, super keyword is used to refer to the parent class of a subclass. Here are some of its key characteristics:

**1**.**super is used to call a superclass constructor**—When a subclass is created, its constructor must call the constructor of its parent class. This is done using the super() keyword, which calls the constructor of the parent class.

**Example**

class Superclass {

int num;

*// Superclass constructor*

Superclass(int *num*) {

this.num = num;

}

void display() {

System.out.println("Number in superclass: " + num);

}

}

class Subclass extends Superclass {

*// Subclass constructor*

Subclass(int *num*) {

super(num); *// Calling superclass constructor*

}

void display() {

System.out.println("Number in subclass: " + num);

}

}

public class Main {

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

Subclass obj = *new* Subclass(10);

obj.display(); *// This will call the display method of Subclass*

}

}

Java

**Output**

Number in subclass*:* 10

Java

**2**. **super is used to call a superclass method:** A subclass can call a method defined in its parent class using the super keyword. This is useful when the subclass wants to invoke the parent class’s implementation of the method in addition to its own.

**Example**

class Superclass {

void display() {

System.out.println("This is the superclass display method");

}

}

class Subclass extends Superclass {

*// Subclass method overriding the superclass method*

void display() {

super.display(); *// Calling superclass method using super keyword*

System.out.println("This is the subclass display method");

}

}

public class Main {

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

Subclass obj = *new* Subclass();

obj.display(); *// This will call the display method of Subclass*

}

}

Java

**Output**

This is the superclass display method

This is the subclass display method

Java

**3.super is used to access a superclass field:**A subclass can access a field defined in its parent class using the super keyword. This is useful when the subclass wants to reference the parent class’s version of a field.

**Example**

class Superclass {

int num = 10;

}

class Subclass extends Superclass {

void display() {

System.out.println("Value of num in superclass: " + super.num); *// Accessing superclass field using super keyword*

}

}

public class Main {

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

Subclass obj = *new* Subclass();

obj.display(); *// This will display the value of num from the superclass*

}

}

Java

**Output**

Value of num in superclass*:* 10

Java

**4**. **super must be the first statement in a constructor:**When calling a superclass constructor, the super() statement must be the first statement in the constructor of the subclass.

**Example**

class Superclass {

int num;

*// Superclass constructor*

Superclass(int *num*) {

this.num = num;

}

}

class Subclass extends Superclass {

int anotherNum;

*// Subclass constructor*

Subclass(int *num*, int *anotherNum*) {

super(num); *// Calling superclass constructor - must be the first statement*

this.anotherNum = anotherNum;

}

void display() {

System.out.println("Number in superclass: " + num);

System.out.println("Another number in subclass: " + anotherNum);

}

}

public class Main {

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

Subclass obj = *new* Subclass(10, 20);

obj.display(); *// This will display the numbers from both superclass and subclass*

}

}

Java

**Output**

Number in superclass*:* 10

Another number in subclass*:* 20

Java

**5**.**super cannot be used in a static context:** The super keyword cannot be used in a static context, such as in a static method or a static variable initializer.

**Example**

class Superclass {

static int num = 10;

}

class Subclass extends Superclass {

static void display() {

*// This will result in a compilation error because super cannot be used in a static context*

*// System.out.println("Value of num in superclass: " + super.num);*

}

}

public class Main {

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

Subclass.display(); *// Calling the static method of Subclass*

}

}

Java

**Output**

*// This will result in a compilation error because super cannot be used in a static context*

System.out.println("Value of num in superclass: " + Superclass.num);

Java

**6. super is not required to call a superclass method:**While it is possible to use the super keyword to call a method in the parent class, it is not required. If a method is not overridden in the subclass, then calling it without the super keyword will invoke the parent class’s implementation.

**Example**

class Superclass {

void display() {

System.out.println("This is the superclass display method");

}

}

class Subclass extends Superclass {

*// Subclass method overriding the superclass method*

void display() {

*// Calling superclass method without using super keyword*

super.display();

System.out.println("This is the subclass display method");

}

}

public class Main {

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

Subclass obj = *new* Subclass();

obj.display(); *// This will call the overridden display method of Subclass*

}

}

Java

**Output**

This is the superclass display method

This is the subclass display method

Java

**Use of Super Keyword in Java**

It is majorly used in the following contexts as mentioned below:

* Use of super with Variables
* Use of super with Methods
* Use of super with Constructors

Example

class Superclass {

int num = 10;

void display() {

System.out.println("This is the superclass display method");

}

*// Superclass constructor*

Superclass() {

System.out.println("This is the superclass constructor");

}

}

class Subclass extends Superclass {

int anotherNum = 20;

*// Subclass constructor*

Subclass() {

super(); *// Calling superclass constructor*

System.out.println("This is the subclass constructor");

}

void display() {

super.display(); *// Calling superclass method*

System.out.println("This is the subclass display method");

}

void accessVariables() {

System.out.println("Value of num in superclass: " + super.num); *// Accessing superclass variable using super keyword*

System.out.println("Value of anotherNum in subclass: " + anotherNum);

}

}

public class Main {

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

Subclass obj = *new* Subclass();

obj.display(); *// This will call the display method of Subclass*

obj.accessVariables(); *// This will access the variables from both superclass and subclass*

}

}

Java

Output

This is the superclass constructor

This is the subclass constructor

This is the superclass display method

This is the subclass display method

Value of num in superclass*:* 10

Value of anotherNum in subclass*:* 20

Java

**Advantages of Using Java Super Keyword**

Using the **super** keyword in Java offers several advantages, including:

1. **Accessing superclass members**: Allows accessing members (variables, methods, and constructors) of the superclass from within the subclass.
2. **Method overriding**: Facilitates calling overridden methods of the superclass from within the subclass, enabling the extension of functionality without losing the original behavior.
3. **Constructor chaining**: Enables calling the constructor of the superclass from within the subclass constructor, allowing initialization of inherited members and ensuring proper initialization of the superclass.
4. **Clarity and readability**: Enhances code clarity and readability by explicitly indicating the invocation of superclass members or constructors, improving code comprehension for developers.
5. **Avoiding ambiguity**: Helps to distinguish between superclass and subclass members with the same name, ensuring clarity in cases of shadowing or overriding.
6. **Maintaining code consistency**: Promotes consistent usage of superclass members across subclasses, facilitating code maintenance and reducing the likelihood of errors.

**‘Final’ Keyword in java :**

**Introduction**

While working on a programming language, programmers often have to finalize things that can’t be changed or overridden by someone. They might restrict other classes to inherit classes, limit methods to override, finalize variables, and whatnot.

Programmers use the Final keyword in Java to restrict other entities to override or modify the entity or class declared with the Final keyword.

The Final keyword is a non-access modifier applicable to a method, variable, or class. It restricts entities to modify or override a class. It is used to show that an entity can’t be modified twice. Hence, we can’t override a method, change the value of a variable, or inherit a class once it’s declared with the Final keyword.

Java Final keyword provides more functionality and can be used with classes, [variables](https://www.tutorialsfreak.com/java-tutorial/java-variables), fields, [*methods*](https://www.tutorialsfreak.com/java-tutorial/java-methods), and function parameters. Final variables, final methods, and final classes enable us to restrict variables to re-initialize, methods to override, and classes to inherit, respectively. The Final keyword can be primitive and non-primitive [data types](https://www.tutorialsfreak.com/java-tutorial/java-data-types).

**final**also has uses beyond creating immutable objects. It can also be used to prevent inheritance or to make a class static. In short, the **final** keyword is a versatile tool that can be used in a variety of ways.

The following are**different contexts** where the final is used:

1. Variable
2. Method
3. Class

**Characteristics of Final Keyword in Java**

In Java, the **final** keyword is used to apply restrictions on classes, methods, and variables. Here are its characteristics:

1. **Final Variables**: When applied to a variable, it means that the variable can only be assigned once. Once initialized, its value cannot be changed. This is particularly useful when you want to create constants.
2. **Final Methods**: When applied to a method, it means that the method cannot be overridden by subclasses. This is often used to enforce immutability or security in classes.
3. **Final Classes**: When applied to a class, it means that the class cannot be subclassed. This is often used to prevent modification or extension of certain classes, particularly utility classes.
4. **Memory Semantics**: When a **final** variable is initialized in a constructor, it’s guaranteed to be initialized before any other threads can access the object. This provides a certain level of thread safety in multi-threaded environments.
5. **Compiler Optimization**: The **final** keyword can provide hints to the compiler to perform optimizations. For example, it may inline constant values or methods, improving performance.
6. **Security**: **final** keyword can be used to prevent certain types of attacks such as Man-in-the-middle attacks by ensuring certain methods or variables cannot be modified.

**Java Final Variable**

Programmers can create a final variable when they need to create a constant variable, which means after initializing it cannot be modified or updated.

If we try to modify this variable, the compiler will throw an error. The same way they can do with objects but, in this case we can change the property of objects as we only finalize the reference variable of the object.

Final variables can be used with primitive data types (int, float, double, and char) and non-primitive data types (object references).

**Different Methods of Using Final Variable**

**Final Variables as Constants**

Declare a **final** variable and initialize it with a constant value. Once you initialized the value, its value cannot be changed.

public class FinalExample {

public static final int MAX\_VALUE = 100;

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

*// Access the final variable*

System.out.println("Maximum value: " + MAX\_VALUE);

}

}

Java

**Output**

Maximum value*:* 100

Java

**Final Variables in Methods**

Declare method parameters as **final** to indicate that the method cannot change their values.

public class FinalMethodExample {

public void printNumber(final int *num*) {

*// num = 10; // This will cause a compilation error*

System.out.println("Number: " + num);

}

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

FinalMethodExample example = *new* FinalMethodExample();

example.printNumber(5);

}

}

Java

**Output**

Number*:* 5

Java

**Final Variables in Classes**

Declare a class as **final** to indicate that it cannot be subclassed.

final class FinalClass {

*// Class definition*

}

*// This will cause a compilation error since FinalClass cannot be extended*

*// class SubClass extends FinalClass { }*

Java

**Output**

The code defines a final class named FinalClass, indicating that it cannot be subclassed.

Then, it attempts to create a subclass SubClass extending FinalClass, which will result

in a compilation error due to the attempt to extend a final class.

Java

**Final Variables in Instance Fields**

Declare instance fields as **final** to ensure they can only be initialized once, either during declaration or in the constructor.

public class FinalInstanceFieldExample {

private final int value;

public FinalInstanceFieldExample(int *value*) {

this.value = value;

}

public void printValue() {

System.out.println("Value: " + value);

}

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

FinalInstanceFieldExample example = *new* FinalInstanceFieldExample(10);

example.printValue();

}

}

Java

**Output**

Value*:* 10

Java

**Java Final classes**

In Java, a final class prevents subclassing. Declaring a class as final means no other class can inherit from it. This ensures that a particular class, often representing a fundamental concept or functionality, cannot be extended or modified.

**Example**

final class FinalClass {

private final int value;

public FinalClass(int *value*) {

this.value = value;

}

public int getValue() {

*return* value;

}

}

Java

**Output**

This code defines a final class named FinalClass with a private final instance field value

of type int. The constructor initializes this field with the provided value. There's also

a method getValue() that allows access to the value of this field.

Since there's no main method, the code doesn't have an output by itself. It just defines

a class that can be used elsewhere. If you want to see the output, you'd need to use this

class in another class with a main method and invoke its methods.

Java

**Java Final Method**

In Java, a final method cannot be overridden by subclasses. When declared as final in a superclass, it means subclasses cannot alter its implementation.

**Example**

class ParentClass {

*// Final method*

public final void finalMethod() {

System.out.println("This is a final method in the ParentClass");

}

}

class ChildClass extends ParentClass {

*// Uncommenting this method will result in a compilation error*

*// @Override*

*// public void finalMethod() {*

*// System.out.println("This is an overridden final method in the ChildClass");*

*// }*

}

public class FinalMethodExample {

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

ParentClass parent = *new* ParentClass();

parent.finalMethod(); *// Output: This is a final method in the ParentClass*

ChildClass child = *new* ChildClass();

child.finalMethod(); *// Output: This is a final method in the ParentClass*

}

}

Java

**Output**

This is a final method in the ParentClass

This is a final method in the ParentClass

Java

In this example, **ParentClass** contains a final method **finalMethod()**. Attempts to override this method in a subclass like **ChildClass** are commented out, as overriding a final method would result in a compilation error. The **FinalMethodExample** class demonstrates calling the final method both through an instance of the parent class and an instance of the child class.

**Advantages of Final Keyword in Java**

The **final** keyword in Java offers several advantages:

1. **Immutability**: When applied to variables, the **final** keyword ensures that the variable’s value cannot be changed after initialization. This promotes immutability, which can lead to safer and more predictable code, especially in concurrent or multithreaded environments.
2. **Security**: Marking classes, methods, or variables as **final** prevents them from being subclassed, overridden, or reassigned, respectively. This can enhance security by preventing unintended modifications or extensions that could compromise the integrity or behavior of the code.
3. **Performance Optimization**: The **final** keyword provides hints to the compiler and runtime environment that certain elements will not change, allowing them to perform optimizations such as inlining or caching. This can lead to improved performance in some cases.
4. **API Design and Contract**: Using **final** can serve as a form of documentation and contract, signaling to other developers that a particular class, method, or variable is intended to be used as-is and should not be modified or extended. This can improve code readability and maintainability.
5. **Thread Safety**: In multithreaded environments, **final** variables can be safely accessed by multiple threads without the risk of race conditions or unexpected changes in value, as their values are guaranteed to be immutable after initialization.
6. **Compiler Checks**: The **final** keyword enables the compiler to perform additional checks and optimizations, such as ensuring that **final** variables are initialized before use or that **final** methods are not overridden incorrectly. This helps catch errors at compile time rather than runtime.

Overall, the **final** keyword in Java provides a powerful mechanism for enhancing code robustness, security, performance, and maintainability by enforcing immutability, preventing unintended modifications, and enabling compiler optimizations.

**Frequently Asked Questions**

**What is the purpose of the final keyword in Java?**

The **final** keyword in Java is used to declare entities – variables, methods, or classes – that cannot be changed or overridden after initialization or declaration.

**Can a final variable be reassigned in Java?**

No, once initialized, the value of a **final** variable cannot be changed. Attempts to reassign a value to a **final** variable will result in a compilation error.

**What happens if I try to override a final method in a subclass?**

Attempting to override a **final** method in a subclass will result in a compilation error. **final** methods are meant to be unchangeable and should not be overridden.

**Can I subclass a final class in Java?**

No, a **final** class cannot be subclassed. Attempting to extend a **final** class will lead to a compilation error.

**Why would I use the final keyword in Java?**

Using the **final** keyword promotes immutability, enhances security by preventing unintended modifications, facilitates performance optimizations, and serves as documentation for API contracts.

* **Garbage Collection :**

**Garbage Collection in Java**

Garbage Collection (GC) in Java is an automatic process that manages memory by reclaiming unused objects and preventing memory leaks. The Java Virtual Machine (JVM) includes a garbage collector that identifies objects that are no longer accessible and deallocates their memory to make space for new objects.

**Key Points of Garbage Collection:**

1. **Automatic Memory Management:** Java does not require manual memory deallocation like C or C++. The JVM manages memory automatically.
2. **Heap Memory Cleanup:** Objects in Java are stored in the heap memory, and the garbage collector reclaims memory occupied by unreachable objects.
3. **No Guaranteed Execution Time:** The garbage collector runs unpredictably based on JVM decisions, memory availability, and system load.
4. **Prevents Memory Leaks:** By removing unused objects, it ensures efficient memory utilization and avoids memory leaks that could lead to application crashes.

**Requesting Garbage Collection Using gc() Method**

Although the JVM automatically manages garbage collection, developers can **request** it using the gc() method. However, calling this method **does not guarantee** immediate garbage collection. The JVM decides when to run the garbage collector based on its internal algorithms and system resources.

**Ways to Request Garbage Collection:**

* **Using System.gc()** → Requests JVM to run the garbage collector.
* **Using Runtime.getRuntime().gc()** → Calls the garbage collector via the runtime instance of the JVM.

Despite these methods, the JVM may ignore the request if it determines that garbage collection is unnecessary at that moment.

**Ways to Make an Object Eligible for Garbage Collection**

An object becomes **eligible for garbage collection** when there are no more references to it in the program. Here are some ways this can happen:

1. **Nullifying the Reference:**
   * Assigning null to an object reference removes the reference, making the object eligible for garbage collection.
2. **Reassigning a Reference Variable:**
   * If an object reference is reassigned to another object, the previously referenced object becomes unreachable and eligible for GC.
3. **Objects Inside a Method (Local Objects):**
   * Objects created inside a method become unreachable once the method completes execution, as there are no references left to access them.
4. **Isolating Objects (Island of Isolation):**
   * When two objects reference each other but have no external references, they form an "island of isolation" and become eligible for GC.
5. **Weak References:**
   * Objects referenced only through weak references (WeakReference or SoftReference) may be garbage collected when memory is needed.
6. **Explicitly Calling finalize() (Deprecated in Java 9+):**
   * Previously, developers could override the finalize() method, but it is now deprecated due to inefficiency and unpredictability.

**Conclusion**

Garbage collection is an essential part of Java’s memory management system, preventing memory leaks and optimizing performance. While developers can **request** GC, they have limited control over when it actually occurs. Understanding object eligibility for GC helps in writing efficient, memory-conscious Java programs.

* **Instance of Operator :**
* The instance of operator in Java is used to check if an object belongs to a specific class or its subclasses. It returns true if the object is an instance of the specified class or a subclass, and false otherwise.
* class Animal {}
* class Dog extends Animal {}
* public class InstanceOfExample {
* public static void main(String[] *args*) {
* Animal animal = *new* Animal();
* Dog dog = *new* Dog();
* *// Using instanceof to check if an object is an instance of a class*
* System.out.println("Is 'animal' an instance of Animal? " + (animal instanceof Animal)); *// true*
* System.out.println("Is 'animal' an instance of Dog? " + (animal instanceof Dog)); *// false*
* System.out.println("Is 'dog' an instance of Animal? " + (dog instanceof Animal)); *// true*
* System.out.println("Is 'dog' an instance of Dog? " + (dog instanceof Dog)); *// true*
* *// Using instanceof to check if an object is an instance of a superclass*
* System.out.println("Is 'dog' an instance of Object? " + (dog instanceof Object)); *// true*
* *// Using instanceof to check if an object is an instance of an interface*
* System.out.println("Is 'dog' an instance of Animal interface? " + (dog instanceof Animal)); *// true*
* }
* }

**Day 10**

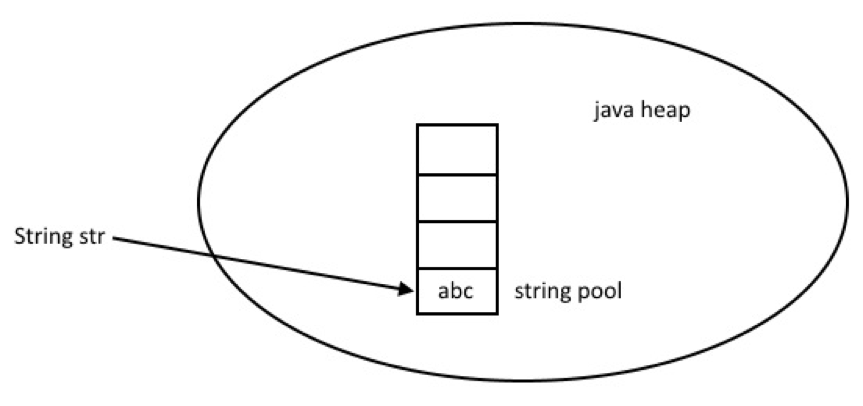
* **String in java:-** Strings are a crucial part of any programming language, including the string of Java. We use Strings to represent and manipulate text-based data, which is a large portion of the data we work with daily.
* The String class is one of the classes which implement this interface. Hence, the string is an object that represents a sequence of char values.
* Strings in Java are unique because, unlike other objects, they are *immutable*. This means that once a String object is created, it cannot be changed. Instead, each time you manipulate a String, Java creates a new String object.
* **Different Ways to Create String**

There are many ways to create a string object in Java, some of the popular ones are given below.

1. **Using string literal** :- This is the most common way of creating strings. In this case, a string literal is enclosed with double quotes.

String str = "abc";

Java



When we create a String using double quotes, JVM looks in the [String pool](https://www.digitalocean.com/community/tutorials/what-is-java-string-pool) to find if any other String is stored with the same value. If found, it just returns the reference to that String object else it creates a new String object with a given value and stores it in the String pool.

2. **Using new keyword**

you can create a string using the new keyword by explicitly invoking the constructor of the String

String str = *new* String("Hello, World!");

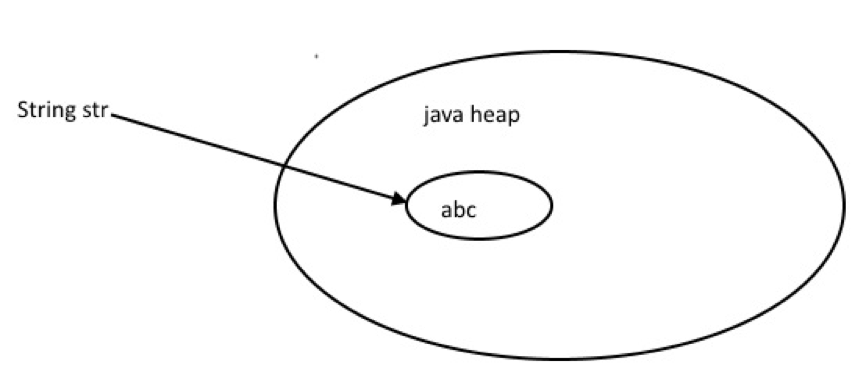
Java

String str = *new* String("abc");

char[] a = {'a', 'b', 'c'};

String str2 = *new* String(a);

Java



**Java String class methods**

**The java.lang.String class provides many useful methods to perform operations on a sequence of char values.**

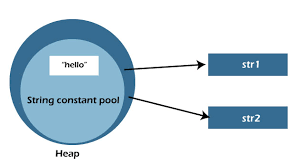
|  |  |  |
| --- | --- | --- |
| Method | Return Type | Use Case |
| charAt() | char | The character at the provided index is returned by charAt(). |
| codePointBefore() | int | Returns the Unicode of the character before the specified index. |
| codePointAt() | int | codePointAt() returns the Unicode of the character at the specified index. |
| compareTo() | int | It compares two strings lexicographically |
| compareToIgnoreCase() | int | It compares two strings lexicographically, ignoring case differences. |
| concat() | String | A string is appended to the end of another string by this function. |
| contains() | boolean | Determines whether a string contains a given character sequence. |
| contentEquals() | boolean | Checks whether a string includes the same character sequence as the provided CharSequence or StringBuffer. |
| copyValueOf() | String | It returns a String containing all of the characters in the character array. |
| endsWith() | boolean | endsWith() determines whether a string contains the provided character at the end. |
| equals() | boolean | This function compares two strings. If the strings are equal, returns true; otherwise, returns false. |
| equalsIgnoreCase() | boolean | equalsIgnoreCase() compares two strings without taking case into account. |
| hashCode() | int | hashCode() returns the hash code of a string. |
| indexOf() | int | In a string, this function returns the position of the first discovered occurrence of provided characters. |
| intern() | String | intern() returns the string object’s canonical representation |
| isEmpty() | boolean | Determines whether or not a string is empty. |
| lastIndexOf() |  | In a string, this function returns the position of the last discovered occurrence of a provided character. |
| length() | int | This function returns the length of the string. |
| replace() | String | replace() looks for a specified value in a string and returns a new string with the provided values replaced. |
| replaceAll() | String | Each substring of this string that satisfies the supplied regular expression is replaced with the specified replacement by replaceAll(). |
| split() | String[] | split() creates an array of substrings from a string |
| startsWith() | boolean | startsWith() determines whether a string begins with the characters supplied. |
| substring() | String | substring() generates a new string that is a substring of the given string. |
| toLowerCase() | String | Converts a string to lowercase letters. |
| toString() | String | Returns the value of a String object. |
| toUpperCase() | String | Converts a string to upper case letters. |
| trim() | String | Removes whitespace from the beginning and end of a string. |
| valueOf() | String | The string representation of the provided value is returned. |

**here’s an example Java code snippet that demonstrates the use of some important methods provided by the String class:**

This code demonstrates:

* length(): Getting the length of the string.
* charAt(int index): Accessing characters at specific indices.
* substring(int beginIndex, int endIndex): Extracting substrings.
* toUpperCase(), toLowerCase(): Converting the case of the string.
* indexOf(String str): Finding the index of a substring.
* startsWith(String prefix), endsWith(String suffix): Checking if the string starts or ends with a specific prefix or suffix.
* split(String regex): Splitting the string based on a delimiter.
* replace(CharSequence target, CharSequence replacement): Replacing characters.
* trim(): Removing leading and trailing whitespace.
* **String Constant Pool**

The string constant pool in Java is a storage area in the heap memory that stores string literals. When a string is created, the JVM checks if the same value exists in the string pool. If it does, the reference to that existing object is returned. Otherwise, a new string object is created and added to the string pool, and its reference is returned.



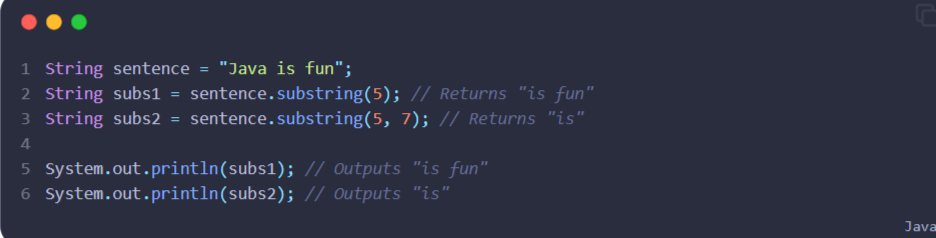
**Here’s an example demonstrating the use of the String constant pool in Java:**

****

1. str1 and str2 are both assigned the same string literal "hello". Since string literals are interned, they will refer to the same object in the string constant pool. Therefore, str1 == str2 evaluates to true.
2. str3 is created using the new keyword, so it will be a separate object from str1 and str2. Therefore, str1 == str3 evaluates to false.
3. By calling the intern() method on str3, we ensure that it is added to the string constant pool if it’s not already present. The reference returned by intern() is then assigned to str4. Since "hello" is already in the pool, str4 will refer to the same object as str1. Therefore, str1 == str4 evaluates to true.

* **What Is A Substring In Java?**
* A substring is a [subset](https://blog.geekster.in/subsets-arrays-java-subsequence-techniques/) of a String. In Java, we can extract a substring from a String using the substring() method. This method comes in two forms:
* substring(int beginIndex): This returns a substring from the specified beginIndex till the end of the string.substring(int beginIndex, int endIndex): This returns a substring from beginIndex (inclusive) to endIndex (exclusive).

**Here’s an example:**

****

* **Java Strings: Mutable or Immutable :-** Although it may seem like string values can be changed in previous sections of this article, the actual string value remains unchanged.
* The concept of the String Constant Pool is closely tied to string immutability in Java.
* In Java, strings are immutable, their values cannot be changed once initialized. This is because a single string object in the String constant pool can have multiple references. Modifying the value of one reference could affect other strings or reference variables, leading to conflicts. To prevent these conflicts, string objects are made immutable in Java.
* In Java, strings are immutable. When we concatenate a string like ” to all” with str, a new string value is created. str then points to this newly created value, while the original value remains unchanged. This behavior demonstrates the immutability of strings in Java.
* **String Comparision**

**String Comparison in Java: ==, equals(), and compareTo()**

In Java, strings can be compared using different methods depending on what exactly you want to check. The three primary ways to compare strings are:

1. **Using == Operator** (Reference Comparison)
2. **Using equals() Method** (Content Comparison)
3. **Using compareTo() Method** (Lexicographical Comparison)

**1. == Operator (Reference Comparison)**

The == operator in Java is used to compare **memory addresses** (references) of two objects. When applied to strings, it checks whether two string variables refer to the same memory location.

**Example:**

java

CopyEdit

public class StringComparison {

public static void main(String[] args) {

String s1 = "Hello";

String s2 = "Hello";

String s3 = new String("Hello");

System.out.println(s1 == s2); // true (same reference in String Pool)

System.out.println(s1 == s3); // false (s3 is a new object in Heap)

}

}

**Explanation:**

* s1 and s2 both refer to the **same object in the String Pool** (Java optimizes memory by reusing string literals).
* s3 is created using new String("Hello"), which explicitly creates a **new object in Heap memory**.
* == checks for **memory location equality**, not the actual content.

**When to use ==?**

* Use == when you want to check whether two references point to the same object, not just equal content.

**2. equals() Method (Content Comparison)**

The equals() method in Java is used to compare the **actual content** (characters) of two strings.

**Example:**

java

CopyEdit

public class StringComparison {

public static void main(String[] args) {

String s1 = "Java";

String s2 = "Java";

String s3 = new String("Java");

System.out.println(s1.equals(s2)); // true (same content)

System.out.println(s1.equals(s3)); // true (same content, different objects)

System.out.println(s1.equals("java")); // false (case-sensitive comparison)

}

}

**Explanation:**

* equals() compares the **actual character sequence** in the string.
* s1 and s2 contain the same content, so equals() returns true.
* s1 and s3 also have the same content, so equals() returns true, even though they are stored in different memory locations.
* String comparison is **case-sensitive**, so "Java" is **not** equal to "java".

**Case-Insensitive Comparison**

To ignore case while comparing, use equalsIgnoreCase().

java

CopyEdit

System.out.println(s1.equalsIgnoreCase("java")); // true

**When to use equals()?**

* Use equals() when you want to check if two strings have the **same content**, regardless of where they are stored in memory.

**3. compareTo() Method (Lexicographical Comparison)**

The compareTo() method is used to compare two strings **lexicographically** (dictionary order). It returns:

* 0 → if both strings are equal.
* A **negative value** → if the first string comes before the second string.
* A **positive value** → if the first string comes after the second string.

**Example:**

java

CopyEdit

public class StringComparison {

public static void main(String[] args) {

String s1 = "Apple";

String s2 = "Banana";

String s3 = "Apple";

System.out.println(s1.compareTo(s2)); // -1 ("Apple" < "Banana")

System.out.println(s2.compareTo(s1)); // 1 ("Banana" > "Apple")

System.out.println(s1.compareTo(s3)); // 0 (Both are "Apple")

}

}

**Explanation:**

* "Apple".compareTo("Banana") → returns -1 because "Apple" is **lexicographically smaller** than "Banana".
* "Banana".compareTo("Apple") → returns 1 because "Banana" is **lexicographically larger** than "Apple".
* "Apple".compareTo("Apple") → returns 0 because both are the same.

**Case Sensitivity**

The compareTo() method is **case-sensitive**, meaning uppercase letters come before lowercase letters in ASCII order.

java

CopyEdit

System.out.println("apple".compareTo("Banana")); // 32 (because lowercase 'a' > uppercase 'B')

**Case-Insensitive Comparison**

To ignore case while comparing:

java

CopyEdit

System.out.println("apple".compareToIgnoreCase("Banana")); // Negative value

**When to use compareTo()?**

* When you need to **sort strings**.
* When you need to determine **lexicographic order** (useful in dictionaries, sorting algorithms, etc.).
* **StringBuilder :**

StringBuilder in Java is an alternative to String class in Java. String class creates immutable string objects which means once a String object is declared, it cannot be modified. However, the StringBuilder class represents a mutable sequence of characters. StringBuilder class is very similar to StringBuffer class. Both these classes are alternative to String class and create mutable character sequences.

However, StringBuilder class operations are faster than StringBuffer because StringBuilder class is not thread-safe. The StringBuilder class provides no guarantee of synchronization, however StringBuffer class operations are synchronized. However, in most cases, operations on a string are performed on the same thread, hence StringBuilder class can be used. StringBuilder class is preferred over StringBuffer class due to its faster operations.

\* Here’s an overview of **StringBuilder** in Java

**1. Instantiation**

You can create a **StringBuilder** object using its constructor in Java:

StringBuilder sb = *new* StringBuilder();

Java

You can also initialize it with an initial value:

StringBuilder sb = *new* StringBuilder("Initial Value");

Java

**2. Methods**

**StringBuilder** provides a variety of methods to manipulate the string it contains:

* **append()**: Appends the string representation of various types of data to the sequence.
* **insert()**: Inserts the string representation of various types of data into the sequence.
* **delete()**: Deletes a sequence of characters from the StringBuilder.
* **deleteCharAt()**: Deletes the character at the specified position.
* **replace()**: Replaces a sequence of characters with another sequence of characters.
* **reverse()**: Reverses the sequence of characters in the StringBuilder.
* **charAt()**: Returns the character at a specified index.
* **indexOf()**: Returns the index within the StringBuilder of the first occurrence of a specified substring.
* **substring()**: Returns a new String that contains a subsequence of characters from the StringBuilder.
* **length()**: Returns the length (number of characters) of the StringBuilder.

**3. Mutability**

Unlike **String**, **StringBuilder** is mutable, meaning you can modify the contents of the string it represents without creating a new object each time. This can be more efficient for tasks that involve a lot of string manipulation.

**4. Performance**

**StringBuilder** is designed for efficiency when performing a series of string manipulations. Because it is mutable, it can avoid the overhead of creating new string objects for each modification. This can lead to better performance, especially for tasks involving a large number of concatenations or modifications.

**When to Use StringBuilder in Java:**

* When you need to concatenate a large number of strings.
* When you need to perform multiple modifications on a string.
* When performance is a concern.

**When Not to Use StringBuilder in Java:**

* When you don’t need mutability or string manipulation. In such cases, using **String** is simpler and more appropriate.
* When the number of string manipulations is very small, the overhead of creating a **StringBuilder** might outweigh the benefits.

Overall, **StringBuilder** is a useful class in Java for efficiently building and manipulating strings

* Below is a simple Java code snippet that demonstrates the usage of **StringBuilder** along with some important methods
* **StringBuffer :**
* The StringBuffer class in java is used for storing the mutable sequence of different datatypes which means we can update the sequence of the StringBuffer class very easily and efficiently without creating a new sequence in memory.
* StringBuffer is faster than the String class and provides various additional methods for deleting the sequence elements, updating the sequence elements, etc. The memory allocation of the StringBuffer object is done in the heap section of memory.
* **StringBuffer Key Points in Java**
* **Mutable**
* Unlike the **String** class, which is immutable (meaning once created, its value cannot be changed), **StringBuffer** allows you to modify the contents of the string without creating a new object each time.
* **Thread-safe**
* The methods in **StringBuffer** are synchronized, making it safe for use in multithreaded environments where multiple threads may try to modify the string concurrently. However, this synchronization can introduce some performance overhead.
* **Performance**
* Because **StringBuffer** is designed for mutable strings, it’s generally more efficient than using concatenation with the **+** operator or **String.concat()** method, especially when dealing with large strings or concatenating within loops.
* **Methods**
* **StringBuffer** provides methods to append, insert, delete, reverse, replace, and manipulate strings in various ways. These methods allow you to modify the content of the **StringBuffer** object.
* **append()**: Adds the specified string representation to the end of the **StringBuffer**.
* **insert()**: Inserts the specified string representation at the specified position.
* **delete()**: Removes a sequence of characters from the **StringBuffer**.
* **reverse()**: Reverses the order of characters in the **StringBuffer**.
* **replace()**: Replaces characters in the **StringBuffer** with new characters.
* Here’s a simple example demonstrating the usage of **StringBuffer**
* public class StringBufferExample {
* public static void main(String[] *args*) {
* StringBuffer sb = *new* StringBuffer("Hello");
* *// Append*
* sb.append(" World");
* System.out.println(sb); *// Output: Hello World*
* *// Insert*
* sb.insert(5, ", ");
* System.out.println(sb); *// Output: Hello, World*
* *// Delete*
* sb.delete(5, 7);
* System.out.println(sb); *// Output: HelloWorld*
* *// Reverse*
* sb.reverse();
* System.out.println(sb); *// Output: dlroWolleH*
* }
* }
* Java
* **Advantages of Using StringBuffer in Java**
* **Mutability**
* **StringBuffer** objects are mutable, meaning you can modify the contents of the string without creating a new object each time. This can lead to better performance and memory utilization, especially when dealing with extensive string manipulation operations.
* **Efficient String Manipulation**
* **StringBuffer** provides efficient methods for string manipulation, such as appending, inserting, deleting, replacing, and reversing characters. These methods allow you to modify the content of the **StringBuffer** object without creating intermediate string objects, which can improve performance, particularly for large strings or concatenations within loops.
* **Thread Safety**
* **StringBuffer** is synchronized, making it safe for use in multithreaded environments where multiple threads may try to modify the string concurrently. This ensures that operations on **StringBuffer** objects are atomic and avoid race conditions. However, this synchronization can introduce some performance overhead.
* **Backward Compatibility**
* **StringBuffer** has been part of the Java language since the early versions, providing backward compatibility for older codebases. Although **StringBuilder** is a newer alternative with better performance in single-threaded scenarios due to its lack of synchronization, **StringBuffer** remains relevant for scenarios requiring thread safety.
* Overall, the advantages of **StringBuffer** make it a suitable choice when you need mutability, efficient string manipulation, and thread safety in your Java applications
* **Comparing String, StringBuilder, and StringBuffer in Java**
* Here is a simple comparison table for String, StringBuilder, and StringBuffer

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **String** | **StringBuilder** | **StringBuffer** |
| **Mutability** | Immutable | Mutable | Mutable |
| **read-Safety** | Immutable (therefore, thread-safe) | Not thread-safe | Thread-safe |
| **Performance** | Slow when string is modified often | Fast | Slower than StringBuilder |
| **Method Availability** | Limited methods for manipulation | Rich set of methods for manipulation | Rich set of methods for manipulation |

**Day 11**

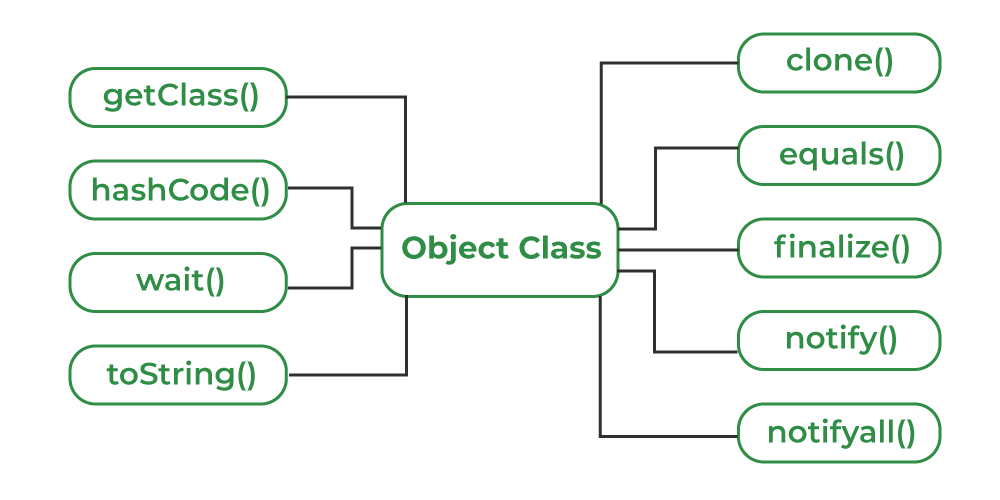
* **Object Class in Java :-**
* **Object class** in Java is present in **java.lang** package. Every class in Java is directly or indirectly derived from the Object class. If a class does not extend any other class then it is a direct child class of the **Java Object class** and if it extends another class then it is indirectly derived. The Object class provides several methods such as **toString(),equals()**, **hashCode(),** and many others. Hence, the Object class acts as a root of the inheritance hierarchy in any Java Program.

**Example:**Here, we will use the [toString()](https://www.geeksforgeeks.org/object-tostring-method-in-java/" \t "_blank) and **hashCode()** methods ***to provide a custom string representation for a class***.

**Object Class Methods**

The Object class provides multiple methods which are as follows:

* toString() method
* hashCode() method
* equals(Object obj) method
* finalize() method
* getClass() method
* clone() method
* wait(), notify() notifyAll() (Concurrency methods)



**1. toString() Method**

The **toString()** provides a String representation of an object and is used to convert an object to a String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@’, and the unsigned hexadecimal representation of the hash code of the object.

**Note:** Whenever we try to print any Object reference, then internally toString() method is called.

**Example:**

*public class Student {*

*public String toString() {*

*return “Student object”;*

*}*

*}*

**Explanation:**The toString() method is overridden to return a custom string representation of the Student object.

**2. hashCode() Method**

For every object, JVM generates a unique number which is a hashcode. It returns distinct integers for distinct objects. A common misconception about this method is that the hashCode() method returns the address of the object, which is not correct. It converts the internal address of the object to an integer by using an algorithm. The hashCode() method is **native** because in Java it is impossible to find the address of an object, so it uses native languages like C/C++ to find the address of the object.

**Use of hashCode() method:**

It returns a hash value that is used to search objects in a collection. JVM(Java Virtual Machine) uses the hashcode method while saving objects into hashing-related data structures like HashSet, HashMap, Hashtable, etc. The main advantage of saving objects based on hash code is that searching becomes easy.

***Note:*** *Override of* ***hashCode()*** *method needs to be done such that for every object we generate a unique number. For example, for a Student class, we can return the roll no. of a student from the hashCode() method as it is unique.*

**Example:**

*public class Student {*

*int roll;*

*@Override*

*public int hashCode() {*

*return roll;*

*}*

*}*

**Explanation:**The **hashCode()** method is overridden to return a custom hash value based on the roll of the Student object.

**3. equals(Object obj) Method**

The **equals()** method compares the given object with the current object. It is recommended to override this method to define custom equality conditions.

**Note:** It is generally necessary to override the **hashCode()** method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

Example:

*public class Student {*

*int roll;*

*@Override*

*public boolean equals(Object o) {*

*if (o instanceof Student) {*

*return this.roll == ((Student) o).roll;*

*}*

*return false;*

*}*

*}*

**Explanation:**The **equals()** method is overridden to compare **roll** between two Student objects.

**4. getClass() method**

The **getClass()** method returns the class object of “this” object and is used to get the actual runtime class of the object. It can also be used to get metadata of this class. The returned Class object is the object that is locked by static synchronized methods of the represented class. As it is final so we don’t override it.

**Example:**

*// Demonstrate working of getClass()*

**public** **class** **Geeks** {

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

{

Object o = **new** String("GeeksForGeeks");

Class c = o.getClass();

System.out.println("Class of Object o is: "

+ c.getName());

}

}

**Output**

Class of Object o is: java.lang.String

**Explanation:**The **getClass()** method is used to print the runtime class of the “o” object.

***Note:*** *After loading a .class file, JVM will create an object of the type java.lang.Class in the Heap area. We can use this class object to get Class level information. It is widely used in* [*Reflection*](https://www.geeksforgeeks.org/reflection-in-java/)

**5. finalize() method**

The [finalize()](https://www.geeksforgeeks.org/finalize-method-in-java-and-how-to-override-it/)method is called just before an object is garbage collected. It is called the [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/) on an object when the garbage collector determines that there are no more references to the object. We should override finalize() method to dispose of system resources, perform clean-up activities and minimize memory leaks. For example, before destroying the Servlet objects web container, always called finalize method to perform clean-up activities of the session.

***Note:*** *The finalize method is called just* ***once*** *on an object even though that object is eligible for garbage collection multiple times.*

**Example:**

*// Demonstrate working of finalize()*

**public** **class** **Geeks** {

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

Geeks t = **new** Geeks();

System.out.println(t.hashCode());

t = **null**;

*// calling garbage collector*

System.gc();

System.out.println("end");

}

@Override **protected** void finalize()

{

System.out.println("finalize method called");

}

}

**Output**

1510467688

end

finalize method called

**Explanation:**The **finalize()** method is called just before the object is garbage collected.

**6. clone() method**

The [clone()](https://www.geeksforgeeks.org/clone-method-in-java-2/)method creates and returns a new object that is a copy of the current object.

**Example:**

*public class Book implements Cloneable {*

*private String t; //title*

*public Book(String t) {*

*this.t = t;*

*}*

*@Override*

*public Object clone() throws CloneNotSupportedException {*

*return super.clone();*

*}*

*}*

**Explanation:** The **clone()** method is overridden to return a cloned copy of the **Book object**.

**7. Concurrency Methods: wait(), notify(), and notifyAll()**

These methods are related to [thread Communication in Java](https://www.geeksforgeeks.org/inter-thread-communication-java/). They are used to make threads wait or notify others in concurrent programming.

**Example of using all the Object Class Methods in Java**

**import** **java.io.\***;

**public** **class** **Book** **implements** Cloneable {

**private** String t; *// title*

**private** String a; *// author*

**private** int y; *// year*

**public** Book(String t, String a, int y)

{

**this**.t = t;

**this**.a = a;

**this**.y = y;

}

*// Override the toString method*

@Override **public** String toString()

{

**return** t + " by " + a + " (" + y + ")";

}

*// Override the equals method*

@Override **public** boolean equals(Object o)

{

**if** (o == **null** || !(o **instanceof** Book)) {

**return** **false**;

}

Book other = (Book)o;

**return** **this**.t.equals(other.getTitle())

&& **this**.a.equals(other.getAuthor())

&& **this**.y == other.getYear();

}

*// Override the hashCode method*

@Override **public** int hashCode()

{

int res = 17;

res = 31 \* res + t.hashCode();

res = 31 \* res + a.hashCode();

res = 31 \* res + y;

**return** res;

}

*// Override the clone method*

@Override **public** Book clone()

{

**try** {

**return** (Book)**super**.clone();

}

**catch** (CloneNotSupportedException e) {

**throw** **new** AssertionError();

}

}

*// Override the finalize method*

@Override **protected** void finalize() **throws** Throwable

{

System.out.println("Finalizing " + **this**);

}

**public** String getTitle() { **return** t; }

**public** String getAuthor() { **return** a; }

**public** int getYear() { **return** y; }

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

{

*// Create a Book object and print its details*

Book b1 = **new** Book(

"The Hitchhiker's Guide to the Galaxy",

"Douglas Adams", 1979);

System.out.println(b1);

*// Create a clone of the Book object and print its*

*// details*

Book b2 = b1.clone();

System.out.println(b2);

*// Check if the two objects are equal*

System.out.println("b1 equals b2: "

+ b1.equals(b2));

*// Get the hash code of the two objects*

System.out.println("b1 hash code: "

+ b1.hashCode());

System.out.println("b2 hash code: "

+ b2.hashCode());

*// Set book1 to null to trigger garbage collection*

*// and finalize method*

b1 = **null**;

System.gc();

}

* **Wrapper Class In Java :**

The concept of a java wrapper classes has been a cornerstone of my development work. These classes, designed to convert Java’s primitive data types into reference types, have been instrumental in enabling my projects to leverage the full power of object-oriented programming. From enhancing the functionality of collections to allowing null values and providing utility methods, wrapper classes have enriched my coding experience significantly.

**What is a Wrapper Classes in Java?**

Java is an object-oriented language that converts a primitive data type into a class object; hence, wrapper class objects enable us to convert the original passed value. These wrapper classes support the multithreading and synchronization process. With the help of this, we can work with collections like Vector, LinkedList, and ArrayList.

Java programming language offers java.lang package that includes the Object and Class. Along with the above overview on what is wrapper class in Java, you must know what they represent. Java wrapper classes represent or wrap the primitive data types’ values as an object. When an object is defined in a wrapper class, it includes a field that can store the primitive data types’ values.

**Note – T**hat the object of one data type includes a field of the specific data type only. So, an object’s double type contains the double type of the field only. It represents that value to store the corresponding reference in a reference type’s variable.

When a wrapper class object is created, the space is allocated in the memory. This is where the primitive data type is saved. Moreover, the wrapper class supports some functionalities for transforming the object into primitive data and, eventually, primitive data into the object. These conversion processes happen automatically.

**Process Flow of the Wrapper Class**

In a wrapper class, we create the object with fields or properties, where we can use and store the primitive data types.



Java implements in-built classes corresponding to specific primitive types that can be applied to modify these value types in object types. We can consider and identify these inbuilt classes as wrapper classes or primitive wrapper classes.

**Need of Wrapper Classes**

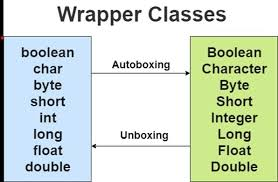
Wrapper classes serve several important purposes in programming, particularly in languages like Java where primitive types cannot be directly used in certain situations. Here’s a list of their key roles and benefits:

* **Object-oriented compatibility**: In object-oriented programming, everything is treated as an object. Wrapper classes allow primitive data types to be treated as objects, enabling them to participate in object-oriented paradigms like inheritance, polymorphism, and encapsulation.
* **Nullable values**: Primitive types cannot represent null values, which can be essential in certain scenarios, such as when dealing with databases or APIs. Wrapper classes like **Integer**, **Double**, etc., allow for nullable values by providing a **null** reference.
* **Collections**: Many collection classes in Java, such as **ArrayList** or **HashMap**, require objects, not primitives. Wrapper classes allow primitive values to be stored in collections by wrapping them in objects.
* **Generics**: Java’s generics do not support primitive types. Therefore, when you need to use generics with primitives, wrapper classes are necessary. For instance, **List<Integer>** is valid, but **List<int>** is not.
* **Type conversions**: Wrapper classes provide methods to convert between primitive types and objects. For example, **Integer.parseInt()** converts a string to an **int**, and **Integer.toString()** converts an **Integer** object to a string.
* **Utility methods**: Wrapper classes offer various utility methods for working with primitive values, such as comparing, sorting, and finding the minimum or maximum value.
* **Interoperability**: In scenarios where primitive types need to be passed to methods expecting objects, wrapper classes facilitate interoperability by allowing primitives to be wrapped and passed as arguments.
* **Constants and methods**: Wrapper classes often provide constants (like **Integer.MAX\_VALUE**) and methods (like **Integer.valueOf()**) that are useful in various programming scenarios.
* **Reflection**: Wrapper classes play a crucial role in reflection, allowing you to inspect and manipulate primitive types as objects during runtime.

**Features of Java Wrapper Classes**

1. Wrapper classes transmute numeric strings into numeric values.
2. We can store primitive data in an object.
3. We can use the valueOf() method in all the wrapper classes without strings.
4. We can use the typeValue() for all the available wrapper classes. It performs as the primitive type and returns the value of the object.

**Primitive Data Types and their Corresponding Wrapper Class**



**Autoboxing and Unboxing**

Autoboxing and unboxing are features introduced in Java 5 to automatically convert between primitive data types and their corresponding wrapper classes. These features simplify code and make it more readable by reducing the need for manual conversion between primitives and wrappers. Here’s how they work:

1. **Autoboxing**:
   * Autoboxing is the automatic conversion of primitive types to their corresponding wrapper classes.
   * For example, when assigning a primitive value to a wrapper class object, autoboxing automatically converts the primitive to its wrapper class equivalent.
2. **Unboxing**:
   * Unboxing is the automatic conversion of wrapper class objects to their corresponding primitive types.
   * For example, when a wrapper class object is used in a context where a primitive type is expected, unboxing automatically extracts the primitive value from the wrapper object.

**Example**

public class AutoboxingUnboxingExample {

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

*// Autoboxing: converting int to Integer*

Integer intValue = 10; *// Autoboxing - int to Integer*

*// Unboxing: converting Integer to int*

int primitiveValue = intValue; *// Unboxing - Integer to int*

System.out.println("Autoboxing: int to Integer - intValue: " + intValue);

System.out.println("Unboxing: Integer to int - primitiveValue: " + primitiveValue);

*// Autoboxing and unboxing in method parameters*

calculateSquare(intValue); *// Autoboxing - int to Integer*

}

public static void calculateSquare(Integer *num*) {

*// Unboxing - Integer to int*

int square = num \* num;

System.out.println("Square of " + num + " is: " + square);

}

}

Java

**Output**

Autoboxing*:* int to Integer - intValue*:* 10

Unboxing*:* Integer to int - primitiveValue*:* 10

Square of 10 is*:* 100

Java

**Explanation:**

* The first **System.out.println** statement prints the value of **intValue**, which is **10**.
* The second **System.out.println** statement prints the value of **primitiveValue**, which is also **10** (after unboxing).
* The **calculateSquare** method is called with **intValue** as an argument. Inside the method, the **Integer** value is unboxed to an **int** and squared. The result, **100**is printed along with the original value **10**.

**Java Wrapper Classes Example**

public class WrapperClassesExample {

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

*// Creating objects of wrapper classes*

Integer intObj = *new* Integer(100); *// Using constructor*

Double doubleObj = Double.valueOf(3.14); *// Using valueOf() method*

Boolean boolObj = Boolean.TRUE; *// Using predefined constant*

*// Autoboxing: assigning primitive values to wrapper objects*

Integer anotherIntObj = 200; *// Autoboxing - int to Integer*

*// Unboxing: extracting primitive values from wrapper objects*

int intValue = intObj.intValue(); *// Unboxing - Integer to int*

double doubleValue = doubleObj.doubleValue(); *// Unboxing - Double to double*

boolean boolValue = boolObj.booleanValue(); *// Unboxing - Boolean to boolean*

*// Printing values*

System.out.println("Integer Object: " + intObj);

System.out.println("Double Object: " + doubleObj);

System.out.println("Boolean Object: " + boolObj);

System.out.println("Another Integer Object: " + anotherIntObj);

System.out.println("Unboxed Integer Value: " + intValue);

System.out.println("Unboxed Double Value: " + doubleValue);

System.out.println("Unboxed Boolean Value: " + boolValue);

*// Using wrapper classes in collections*

java.util.ArrayList<Integer> arrayList = *new* java.util.ArrayList<>();

arrayList.add(10); *// Autoboxing - int to Integer*

arrayList.add(20); *// Autoboxing - int to Integer*

System.out.println("ArrayList: " + arrayList);

}

}

Java

Output

Integer Object*:* 100

Double Object*:* 3.14

Boolean Object*:* true

Another Integer Object*:* 200

Unboxed Integer Value*:* 100

Unboxed Double Value*:* 3.14

Unboxed Boolean Value*:* true

ArrayList*:* [10, 20]

Java

Explanation:

* **Integer**, **Double**, and **Boolean** wrapper objects are created with values **100**, **3.14**, and **true** respectively.
* Autoboxing is demonstrated when assigning **200** to **anotherIntObj**.
* Unboxing occurs when extracting primitive values from wrapper objects (**intValue**, **doubleValue**, **boolValue**).
* Values of all wrapper objects and unboxed primitive values are printed.
* Wrapper objects are used in an **ArrayList**, and the contents of the **ArrayList** are printed.

**Custom Wrapper Classes in Java**

Custom wrapper classes in Java are classes that wrap around primitive types or other objects, providing additional functionality or behavior as needed. Below is an example of a custom wrapper class that wraps around an **int** value and provides methods for incrementing and decrementing the value.

Example

public class CustomIntWrapper {

private int value;

*// Constructor*

public CustomIntWrapper(int *value*) {

this.value = value;

}

*// Getter method*

public int getValue() {

*return* value;

}

*// Setter method*

public void setValue(int *value*) {

this.value = value;

}

*// Method to increment value by one*

public void increment() {

value++;

}

*// Method to decrement value by one*

public void decrement() {

value--;

}

*// Method to add a specific value*

public void add(int *num*) {

value += num;

}

*// Method to subtract a specific value*

public void subtract(int *num*) {

value -= num;

}

*// toString method for string representation*

@Override

public String toString() {

*return* "CustomIntWrapper{" +

"value=" + value +

'}';

}

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

*// Create an instance of CustomIntWrapper*

CustomIntWrapper wrapper = *new* CustomIntWrapper(10);

*// Print initial value*

System.out.println("Initial Value: " + wrapper);

*// Perform operations*

wrapper.increment();

System.out.println("After increment: " + wrapper);

wrapper.add(5);

System.out.println("After adding 5: " + wrapper);

wrapper.subtract(3);

System.out.println("After subtracting 3: " + wrapper);

}

}

Java

Output

Initial Value*:* CustomIntWrapper{value=10}

After increment*:* CustomIntWrapper{value=11}

After adding 5*:* CustomIntWrapper{value=16}

After subtracting 3*:* CustomIntWrapper{value=13}

Java

Explanation:

* Initially, the **CustomIntWrapper** object is created with a value of **10**.
* After each operation (**increment**, **add**, **subtract**), the value of the **CustomIntWrapper** object is updated accordingly, and the object’s string representation is printed.

**Advantages and Disadvantages of Wrapper Class in Java**

**Advantages**

* **Object-oriented compatibility:** Wrapper classes allow primitive data types to be treated as objects, facilitating their use in object-oriented paradigms like inheritance and polymorphism.
* **Nullable values:** Wrapper classes can represent **null** values, which is not possible with primitive types alone. This is useful in scenarios where nullability needs to be explicitly handled, such as database operations.
* **Generics support:** Java generics do not support primitive types, so wrapper classes are necessary when using generics with primitives. For example, **List<Integer>** is valid, but **List<int>** is not.
* **Collections:** Many collection classes in Java (e.g., **ArrayList**, **HashMap**) require objects, not primitives. Wrapper classes allow primitive values to be stored in collections by wrapping them in objects.
* **Utility methods:** Wrapper classes offer utility methods for working with primitive values, such as comparing, sorting, and converting between types.
* **Interoperability:** Wrapper classes enable interoperability by allowing primitives to be wrapped and passed as arguments to methods expecting objects.

**Disadvantages**

* **Performance overhead:** Wrapper classes can introduce performance overhead due to the additional memory consumption and method invocation overhead associated with working with objects.
* **Memory consumption:** Wrapper objects typically consume more memory than their primitive counterparts, as they carry additional metadata and overhead.
* **Autoboxing/unboxing overhead:** Autoboxing (converting primitive types to wrapper objects) and unboxing (extracting primitive values from wrapper objects) can lead to overhead in terms of CPU cycles and memory usage, especially in performance-sensitive applications.
* **Potential for NullPointerException:** When using wrapper classes, there is a risk of encountering **NullPointerException** if proper null checks are not performed, especially when working with autoboxing or methods that return wrapper objects.
* **Complexity:** Introducing wrapper classes can sometimes add complexity to the code, especially in scenarios where autoboxing/unboxing is mixed with manual conversion between primitive types and wrapper objects.

**Methods of Wrapper Class:-**

In Java wrapper classes, valueOf() creates a wrapper object (e.g., Integer, Double) from a primitive value or string, parseInt() parses a string and returns a primitive integer, and xxxValue() methods (e.g., intValue(), doubleValue()) extract the primitive value from a wrapper object.

1. valueOf()

* **Purpose:** Creates a wrapper object (e.g., Integer, Double) from a primitive value or a string representation of that value.
* **Example:**
  + Integer.valueOf(10) creates an Integer object representing the value 10.
  + Integer.valueOf("10") creates an Integer object from the string "10".
* **Return Type:** Wrapper object (e.g., Integer, Double).

2. parseInt()

* **Purpose:** Parses a string and returns the corresponding primitive integer value.
* **Example:**
  + Integer.parseInt("10") returns the primitive integer value 10.
* **Return Type:** Primitive integer (int).

3. xxxValue()

* **Purpose:** Extracts the primitive value from a wrapper object.
* **Example:**
  + Integer.intValue() returns the primitive integer value from an Integer object.
* **Return Type:** Primitive value (e.g., int, double).

**Exception Handling :**

**Introduction**

Exception handling is indeed a fundamental aspect of Java programming, crucial for maintaining the stability and reliability of applications. By effectively managing exceptions, developers can ensure that their programs gracefully handle unexpected events without crashing. Whether dealing with runtime or compile-time exceptions, robust exception handling mechanisms are necessary to safeguard the normal flow of instructions.

In Java, Exception handling serves as indicators of abnormal conditions during program execution, signaling that something undesirable has occurred. These events disrupt the usual flow of execution and require special handling to either recover from the error or gracefully terminate the program. Without proper exception handling, even a minor exception can lead to program crashes, potentially causing inconvenience or data loss to users.

Exception handling isn’t merely about preventing crashes; it also contributes to the overall user experience by making applications more user-friendly. By providing informative error messages and alternative workflows, developers can guide users through unexpected situations, enhancing usability and user satisfaction.

In summary, Exception handling plays a pivotal role in Java programming, with its significance extending beyond mere error prevention. By incorporating robust exception handling strategies, developers can ensure the resilience and user-friendliness of their applications, thereby enhancing the overall quality of the software.

**What are Java Exceptions?**

In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program’s instructions. The program can catch and handle exceptions When an exception occurs within a method, it creates an object. The object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

**Major reasons why an exception Occurs**

* Invalid user input
* Device failure
* Loss of network connection
* Physical limitations (out-of-disk memory)
* Code errors
* Opening an unavailable file

**What are Java Errors?**

Errors in Java represent irrecoverable conditions such as the Java Virtual Machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, and infinite recursion. These errors are typically beyond the control of the programmer and can lead to program termination. It’s generally not advisable to handle errors programmatically, as they often indicate severe issues that require attention at a system level.

**Difference between Error and Exception**

Here are the differences between Errors and Exceptions in Java:

**Errors**

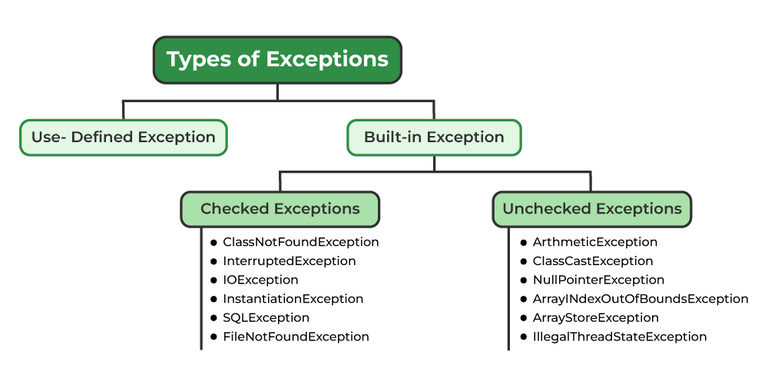
* Errors represent serious, unrecoverable problems that typically result from system-level issues or resource exhaustion.
* The application code generally does not handle errors, as they are beyond its control
* Examples of errors include **OutOfMemoryError**, **StackOverflowError**, and **VirtualMachineError**.
* Errors are unchecked and are subclasses of the **Error** class, which extends **Throwable**.

**Exceptions**

* Exceptions represent conditions that can occur during the normal execution of a program and can be handled by application code.
* Application code handles exceptions, which represent conditions that can occur during the normal execution of a program.
* Examples of exceptions include **NullPointerException**, **ArrayIndexOutOfBoundsException**, and **ArithmeticException**.
* Exceptions are subclasses of the **Exception** class, which extends **Throwable**.

In summary, the main difference lies in the severity and recoverability of the issues they represent.

**Types of Exceptions**



**Methods to print the Exception information in java**

**1. Using print Stack Trace ()**

The print Stack Trace () method of the Throwable class prints the stack trace of the exception, including the class names, method names, and line numbers where the exception occurred.

**Example**

*try* {

*// Code that might throw an exception*

} *catch* (Exception *e*) {

e.printStackTrace();

}

Java

**2. Using getMessage()**

The getMessage() method of the Throwable class returns a string describing the exception.

**Example**

*try* {

*// Code that might throw an exception*

} *catch* (Exception *e*) {

System.out.println("Exception message: " + e.getMessage());

}

Java

**3.Using toString()**

The toString() method of the Throwable class returns a string representation of the exception, which typically includes the class name followed by the message.

**Example**

*try* {

*// Code that might throw an exception*

} *catch* (Exception *e*) {

System.out.println("Exception: " + e.toString());

}

Java

**4. Using logging frameworks**

Java provides logging frameworks like Log4j or java.util.logging. These frameworks offer more sophisticated logging capabilities, allowing you to log exception information with different levels of severity.

**Example**

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

public class MyClass {

private static final Logger logger = LogManager.getLogger(MyClass.class);

public void myMethod() {

*try* {

*// Code that might throw an exception*

} *catch* (Exception *e*) {

logger.error("An exception occurred: ", e);

}

}

}

Java

In this example, the exception and its stack trace are logged at the error level using Log4j.

**5. Using custom logging**

You can also handle exception information manually and log it using custom logging mechanisms like writing to a file or printing to the console.

**Example**

*try* {

*// Code that might throw an exception*

} *catch* (Exception *e*) {

System.err.println("An exception occurred: " + e);

e.printStackTrace(System.err);

}

Java

These are some common methods to print exception information in Java. The choice of method depends on factors such as the desired level of detail, the logging framework being used, and the specific requirements of the application.

**How Does JVM Handle an Exception Handling in Java?**

When an exception occurs in a Java program, the Java Virtual Machine (JVM) takes several steps to handle it:

1. **Exception Thrown:** During the execution of a Java program, when an exceptional condition occurs, such as an arithmetic error or an attempt to access an array element out of bounds, the program creates an exception object representing the specific type of exception.
2. **Exception Propagation:** The JVM searches for the nearest enclosing try-catch block that matches the type of the thrown exception. If such a block is found, the exception is caught, and the corresponding catch block is executed. If not, the exception propagates up the call stack to the caller’s context, searching for an appropriate exception handler.
3. **Stack Unwinding:** As the exception propagates up the call stack, the JVM unwinds the stack, meaning it cleans up the method call frames until it finds a matching catch block or reaches the top level of the call stack. This process releases resources and performs necessary cleanup operations.
4. **Exception Handling:** If a matching catch block is found, the code within that block is executed to handle the exception. The catch block typically contains code to log the exception, perform error recovery, or provide feedback to the user.
5. **Optional Finally Block:** When a try block has an associated finally block, the program executes the code within the finally block regardless of whether an exception occurred or not. This ensures that cleanup operations, such as closing files or releasing resources, are performed even in the presence of exceptions.

Overall, the JVM’s exception handling mechanism ensures that it properly detects, propagates, and handles exceptions, thereby helping to maintain the stability and reliability of Java applications

**Frequently Asked Questions**

**1. What is an exception in Java?**

An exception in Java is an unwanted or unexpected event that occurs during the execution of a program and disrupts its normal flow.

**2. How do I handle exceptions in Java?**

In Java, you handle exceptions using try-catch blocks. The code that might throw an exception is placed inside a try block, and the catch block contains code to handle the exception.

**3. What is the difference between checked and unchecked exceptions?**

In Java, checked exceptions are those that a method must either catch or declare in its signature, while unchecked exceptions (runtime exceptions) do not require explicit handling

* **Try Catch Block :**

**Introduction**

In Java, a try-catch block is a fundamental construct used for handling exceptions gracefully during program execution. It allows you to enclose code that may potentially throw an exception within a try block and provide corresponding handling logic in associated catch blocks. This mechanism ensures that if an exception occurs, the program can gracefully recover or handle the exceptional situation without terminating abruptly.

**Java try block**

In Java, the try block encapsulates a section of code enclosed within the try keyword. It defines a block of code where exceptions might occur. Within this block, developers place the code that could potentially throw an exception. One or more catch blocks or a finally block, or both, must follow the try block.

**Syntax**

*try* {

*// Code that may throw an exception*

}

Java

If an exception occurs during the execution of the code within the try block, it disrupts the normal flow of execution and transfers control to the appropriate catch block (if present) to handle the exception. If no catch block is capable of handling the exception, the program terminates with an uncaught exception.

**Java catch block**

In Java, a catch block is used to handle exceptions that are thrown within a corresponding try block. It is defined using the catch keyword followed by a parameter enclosed in parentheses. This parameter specifies the type of exception that the catch block can handle.

**Syntax**

*try* {

*// Code that may throw an exception*

} *catch* (ExceptionType *e*) {

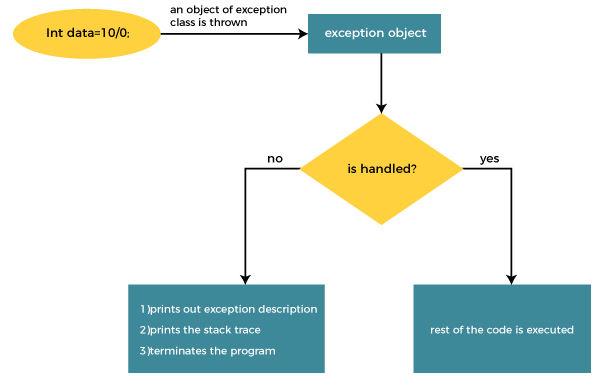
*// Handling logic for the exception*

}

Java

* **ExceptionType**: Specifies the type of exception that the catch block can handle. It can be a specific exception type, such as **ArithmeticException**, **NullPointerException**, or a superclass of exceptions like **Exception** or **Throwable**.
* **e**: This is the exception parameter that represents the exception object thrown during the execution of the try block. You can use this parameter to access information about the exception, such as its message or stack trace.

**Internal Working of Java try-catch block**



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.

**Problem Without Exception Handling**

Example

public class NoExceptionHandlingExample {

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

int result = divide(10, 0); *// Division by zero will cause an ArithmeticException*

System.out.println("Result: " + result); *// This line won't execute if an exception occurs*

}

*// A method that performs division without exception handling*

public static int divide(int *numerator*, int *denominator*) {

*return* numerator / denominator;

}

}

Java

Output

Exception in thread "main" java.lang.ArithmeticException*:* / by zero

at NoExceptionHandlingExample.divide(NoExceptionHandlingExample.java*:*9)

at NoExceptionHandlingExample.main(NoExceptionHandlingExample.java*:*4)

Java

* In this code, we attempt to divide 10 by 0 in the **main** method, which would cause an **ArithmeticException** due to division by zero.
* Since there’s no exception handling mechanism (like try-catch blocks) in place, when the exception occurs, it propagates up the call stack and eventually leads to program termination.
* As a result, the line trying to print the result won’t execute, and instead, an exception stack trace is printed to the console.

**Solution by Exception Handling**

Example

public class ExceptionHandlingExample {

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

*try* {

int result = divide(10, 0); *// Division by zero will cause an ArithmeticException*

System.out.println("Result: " + result); *// This line won't execute if an exception occurs*

} *catch* (ArithmeticException *e*) {

System.out.println("Exception caught: Division by zero!"); *// Handling the ArithmeticException*

e.printStackTrace(); *// Printing the stack trace for more information*

}

}

*// A method that performs division and may throw ArithmeticException*

public static int divide(int *numerator*, int *denominator*) {

*return* numerator / denominator;

}

}

Java

Output

Exception caught*:* Division by zero!

java.lang.ArithmeticException*:* / by zero

at ExceptionHandlingExample.divide(ExceptionHandlingExample.java*:*14)

at ExceptionHandlingExample.main(ExceptionHandlingExample.java*:*6)

Java

* In this code, we enclose the potentially problematic division operation within a try block.
* If an **ArithmeticException** occurs during the execution of the try block (due to division by zero), the control is transferred to the catch block.
* Inside the catch block, we handle the exception by printing a message indicating the exception and printing the stack trace using **e.printStackTrace()** for more detailed information.
* Despite the exception, the program continues to execute after the catch block, allowing any remaining code to run gracefully.

**Conclusion**

In conclusion, the try-catch block in Java provides a powerful mechanism for handling exceptions gracefully during program execution. Key points to remember about try-catch blocks include:

1. **Error Handling**: It allows you to enclose code that may potentially throw an exception within a try block.
2. **Exception Handling**: When an exception occurs within the try block, the control transfers to the corresponding catch block, allowing graceful handling of the exception.
3. **Multiple Catch Blocks**: You can have multiple catch blocks to handle different types of exceptions or handle exceptions at different levels of granularity.
4. **Control Flow**: The program is allowed to continue executing after an exception occurs with exception handling using try-catch blocks, provided that the exception is caught and handled appropriately.
5. **Preventing Abrupt Termination**: By catching exceptions, you can prevent the program from terminating abruptly due to unhandled exceptions, which improves the robustness and reliability of the software.

Overall, try-catch blocks are essential for writing robust and fault-tolerant Java programs by providing a structured way to handle unexpected runtime errors and exceptional conditions.

**Frequently Asked Questions**

**1. What is a try-catch block in Java?**

A try-catch block is a Java programming construct used for handling exceptions. It allows you to encapsulate code that might throw an exception within a try block and provides a mechanism to catch and handle any exceptions that occur during the execution of that code.

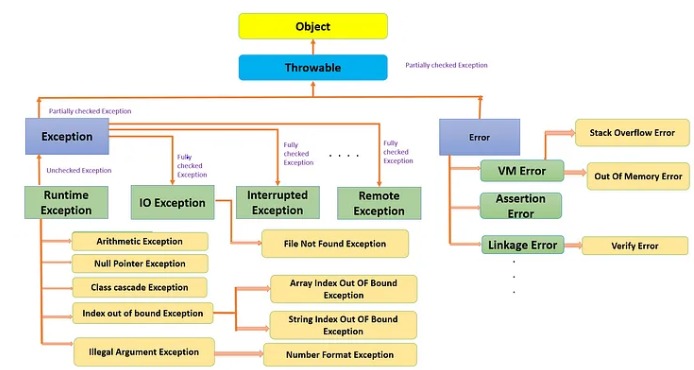
**2. How does a try-catch block work?**

Within a try block, you place the code that might throw an exception. If an exception occurs, control is transferred to the catch block(s) that match the type of exception thrown. The catch block handles the exception by executing code specific to that exception type.

**3. What is the purpose of a catch block?**

The catch block is used to handle exceptions that are thrown within the corresponding try block. It allows you to specify how to handle different types of exceptions, such as printing an error message, logging the exception, or taking corrective action.

**Day 12**

****

* <https://medium.com/@ahmed.abdelfaheem/checked-and-unchecked-exceptions-in-java-6cb1c9815d32>
* **Follow this for checked and unchecked exception**
* **Finally** :- The finally block follows [a try block or a catch block](https://www.tutorialspoint.com/java/java_try_catch_block.htm). A finally block of code always executes, irrespective of occurrence of an Exception.
* Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.
* Throws & Throw :-

**throw and throws in Java :-**

In Java, Exception Handling is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. It handles runtime errors such as **NullPointerException**, **ArrayIndexOutOfBoundsException**, etc. To handle these errors effectively, Java provides two key concepts **throw** and **throws**.

**Difference Between throw and throws**

The main differences between throw and throws in Java are follows:

| Feature | Throw | Throws |
| --- | --- | --- |
| Definition | It is used to explicitly throw an exception. | It is used to declare that a method might throw one or more exceptions. |
| Location | It is used inside a method or a block of code. | It is used in the method signature. |
| Usage | It can throw both checked and unchecked exceptions. | It is only used for checked exceptions. Unchecked exceptions do not require throws. |
| Responsibility | The method or block throws the exception. | The method’s caller is responsible for handling the exception. |
| Flow of Execution | Stops the current flow of execution immediately. | It forces the caller to handle the declared exceptions. |
| Example | throw new ArithmeticException(“Error”); | public void myMethod() throws IOException {} |

* **Try Catch block**
* **Ways to handle checked exception :**

Checked exceptions are the subclass of the Exception class. These types of exceptions need to be handled during the compile time of the program. These exceptions can be handled by the try-catch block or by using throws keyword otherwise the program will give a compilation error.

 ClassNotFoundException, IOException, SQLException etc are the examples of the checked exceptions.

I/O Exception: This Program throws an I/O exception because of due FileNotFoundException is a checked exception in Java. Anytime, we want to read a file from the file system, Java forces us to handle error situations where the file is not present in the given location.

Implementation: Consider GFG.txt file does not exist.

* **System.exit** :- *System.exit* is a *void* method. It takes an exit code, which it passes on to the calling script or program.
* Exiting with a code of **zero means a normal exit:**

Syntax : System.exit(0);

Calling the *System.exit*method terminates the currently running JVM and exits the program. This method does not return normally.

This means that **the subsequent code after the *System.exit* is effectively unreachable** **and yet, the compiler does not know about it.**

* **Propagation of execution :-**
* **What is Exception Propagation?**

Exception propagation in Java refers to the process by which an exception is passed up the call stack from the point where it was thrown to the point where it is caught. This mechanism ensures that exceptions are not only detected but also properly handled by the most appropriate part of the code.

**1.1 How Does Exception Propagation Work?**

* When an exception is thrown, the Java Virtual Machine (JVM) looks for a matching catch block in the current method. If it finds one, the exception is handled there. If not, the JVM propagates the exception to the calling method. This process continues up the call stack until a suitable catch block is found or the program terminates if no such block exists.
* **Methods of unchecked exception :-**
* **printStackCastException**
* **getMessage.**

**Day 13**

* **Packages in Java :-**

**Packages in Java** are a mechanism that encapsulates a group of classes, sub-packages, and interfaces. Packages are used for:

* Prevent **naming conflicts**by allowing classes with the same name to exist in different packages, like **college.staff.cse.Employee** and **college.staff.ee.Employee.**
* They make it easier to organize, locate, and use classes, interfaces, and other components.
* Packages also provide controlled access for Protected members that are accessible within the same package and by subclasses.
* Also for default members (no access specifier) that are accessible only within the same package.

By grouping related classes into packages, Java promotes data encapsulation, making code reusable and easier to manage. Simply import the desired class from a package to use it in your program.

**Working of Java Packages**

**Directory Structure:**Package names and directory structures are closely related. For example, if a package name is **college.staff.cse**, then three directories are, **college, staff, and cse**, where **cse** is inside staff, and staff is inside the college.

**Naming Conventions**: Package names are written in reverse order of domain names, e.g., **org.geeksforgeeks.practice**. In a college, the convention might be:

* college.tech.cse
* college.tech.ee
* college.art.history
* **Access Modifiers in java :-**

**1. Default Access Modifier**

When no access modifier is specified for a class, method, or data member, it is said to be having the default access modifier by default. The default access modifiers are accessible only within the same package.

Example 1: Demonstrating Default Access Modifier Within the Same Package. In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of the second package.

**2. Private Access Modifier**

private access modifier is specified using the keyword private. The methods or data members declared as private are accessible only within the class in which they are declared.

* Any other class of the same package will not be able to access these members.
* Top-level classes or interfaces can not be declared as private because,
  + private The means “only visible within the enclosing class”.
  + protected means “only visible within the enclosing class and any subclasses”.

Hence these modifiers in terms of application to classes, apply only to [nested classes](https://www.geeksforgeeks.org/nested-classes-java/) and not on top-level classes.

**Example**: In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

**3. Protected Access Modifier**

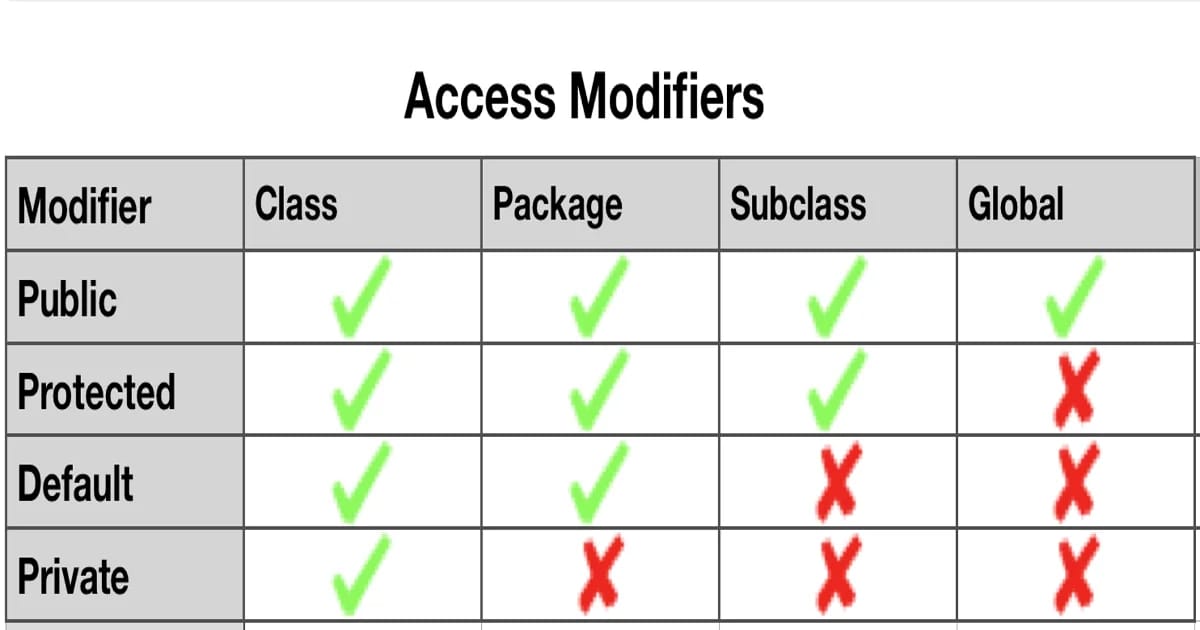
The protected access modifier is specified using the keyword protected. The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

**Example 1**: In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

**4. Public Access Modifier**

The public access modifier is specified using the keyword public.

* The public access modifier has the widest scope among all other access modifiers.
* Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.



**Day 14 : Collection Framework**

**Reffer for basic info** :- <https://blog.geekster.in/java-collections-framework-deep-drive/#:~:text=The%20Java%20Collections%20Framework%20is,process%20groups%20of%20objects%20efficiently>.

**For Detail Reading** : <https://www.geeksforgeeks.org/collections-in-java-2/>

<https://www.tpointtech.com/collections-in-java>

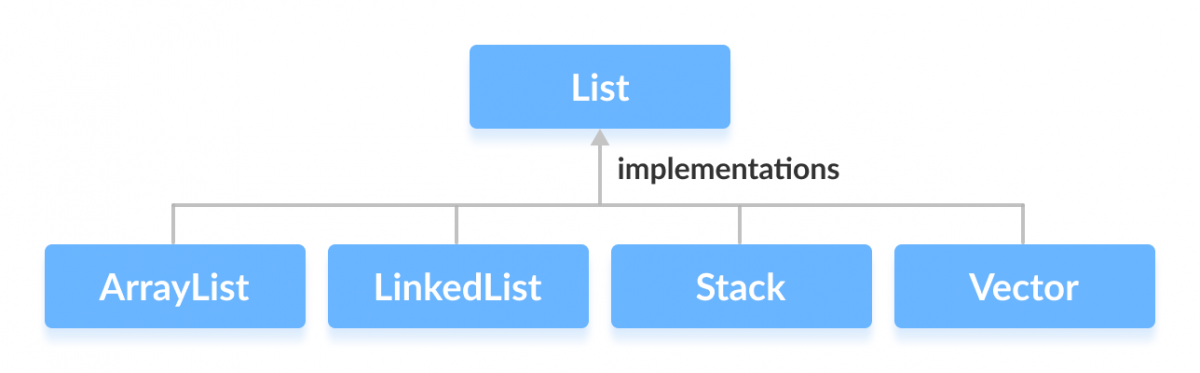
* **Collection Framework :**



**List : Official documentation for list**

[**https://docs.oracle.com/javase/8/docs/api/java/util/List.html**](https://docs.oracle.com/javase/8/docs/api/java/util/List.html)

**Java List :-**



In Java, the List interface is an ordered collection that allows us to store and access elements sequentially. It extends the Collection interface.

**Classes that Implement List**

Since List is an interface, we cannot create objects from it.

In order to use the functionalities of the List interface, we can use these classes:

* ArrayList
* LinkedList
* Vector
* Stack

**Classes implementing the List interface in Java**

These classes are defined in the Collections framework and implement the List interface.

How to use List?

**In Java, we must import java.util.List package in order to use List.**

**// ArrayList implementation of List**

List<String> list1 = new ArrayList<>();

**// LinkedList implementation of List**

List<String> list2 = new LinkedList<>();

Here, we have created objects list1 and list2 of classes ArrayList and LinkedList. These objects can use the functionalities of the List interface.

**Methods of List**

* The List interface includes all the methods of the Collection interface. Its because Collection is a super interface of List.

**Some of the commonly used methods of the Collection interface that's also available in the List interface are:**

**Methods Description**

**add()** adds an element to a list

**addAll()** adds all elements of one list to another

**get()** helps to randomly access elements from lists

**iterator()** returns iterator object that can be used to sequentially access elements of lists

**set()** changes elements of lists

**remove()** removes an element from the list

**removeAll()** removes all the elements from the list

**clear()** removes all the elements from the list (more efficient than removeAll())

**size()** returns the length of lists

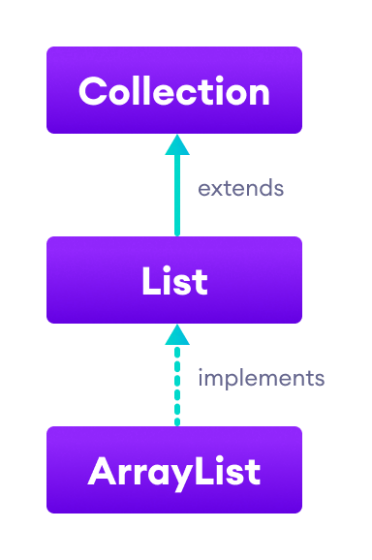
**toArray()** converts a list into an array

contains() returns true if a list contains specific element

* Both the List interface and the Set interface inherit the Collection interface. However, there exists some difference between them.
* Lists can include duplicate elements. However, sets cannot have duplicate elements.

Elements in lists are stored in some order. However, elements in sets are stored in groups like sets in mathematics.

* **ArrayList :-**



* In Java, we use the ArrayList class to implement the functionality of resizable-arrays.
* It implements the List interface of the collections framework .
* The List interface extends the Collection interface and the ArrayList class implements List.

**Java ArrayList Vs Array**

* In Java, we need to declare the size of an array before we can use it. Once the size of an array is declared, it's hard to change it.
* To handle this issue, we can use the ArrayList class. It allows us to create resizable arrays.
* Unlike arrays, arraylists can automatically adjust their capacity when we add or remove elements from them. Hence, arraylists are also known as dynamic arrays.

**Creating an ArrayList**

Before using ArrayList, we need to import the java.util.ArrayList package first. Here is how we can create arraylists in Java:

**ArrayList<Type> arrayList= new ArrayList<>();**

**Here, Type indicates the type of an arraylist. For example,**

**// create Integer type arraylist**

**ArrayList<Integer> arrayList = new ArrayList<>();**

**// create String type arraylist**

**ArrayList<String> arrayList = new ArrayList<>();**

**In the above program, we have used Integer not int. It is because we cannot use primitive types while creating an arraylist. Instead, we have to use the corresponding wrapper classes.**

* **Basic Operations on ArrayList**

The ArrayList class provides various methods to perform different operations on arraylists. We will look at some commonly used arraylist operations in this tutorial:

* **Add elements**
* **Access elements**
* **Change elements**
* **Remove elements**

**1. Add Elements to an ArrayList**

To add a single element to the arraylist, we use the add() method of the ArrayList class.

**2. Access ArrayList Elements**

* To access an element from the arraylist, we use the **get()** method of the ArrayList class.
* We can also access elements of the ArrayList using the iterator() method. To learn more, visit Java ArrayList iterator().

**3. Change ArrayList Elements**

To change elements of the arraylist, we use the set() method of the ArrayList class.

Here, the set() method changes the element at index 2 to JavaScript.

To learn more, visit the Java ArrayList set().

**4. Remove ArrayList Elements**

To remove an element from the arraylist, we can use the remove() method of the ArrayList class. For example,

We can also remove all the elements from the arraylist at once.

**Besides those basic methods, here are some more ArrayList methods that are commonly used.**

**Methods Descriptions**

**size()** Returns the length of the arraylist.

**sort()** Sort the arraylist elements.

**clone(**) Creates a new arraylist with the same element, size, and capacity.

**contains() Searches the arraylist for the specified element and returns a boolean result.**

**ensureCapacity()** Specifies the total element the arraylist can contain.

**isEmpty()** Checks if the arraylist is empty.

**indexOf()** Searches a specified element in an arraylist and returns the index of the **element.**

**Java LinkedList**

* The LinkedList class of the Java collections framework provides the functionality of the linked list data structure (doubly linkedlist).
* A single node of doubly linkedlist that has 3 fields Prev, Data, and Next.

**Java Doubly LinkedList**

**Each element in a linked list is known as a node. It consists of 3 fields:**

**Prev -** stores an address of the previous element in the list. It is null for the first element

**Next** - stores an address of the next element in the list. It is null for the last element

**Data** - stores the actual data

Creating a Java LinkedList

* Here is how we can create linked lists in Java:

**LinkedList<Type> linkedList = new LinkedList<>();**

* Here, Type indicates the type of a linked list. For example,

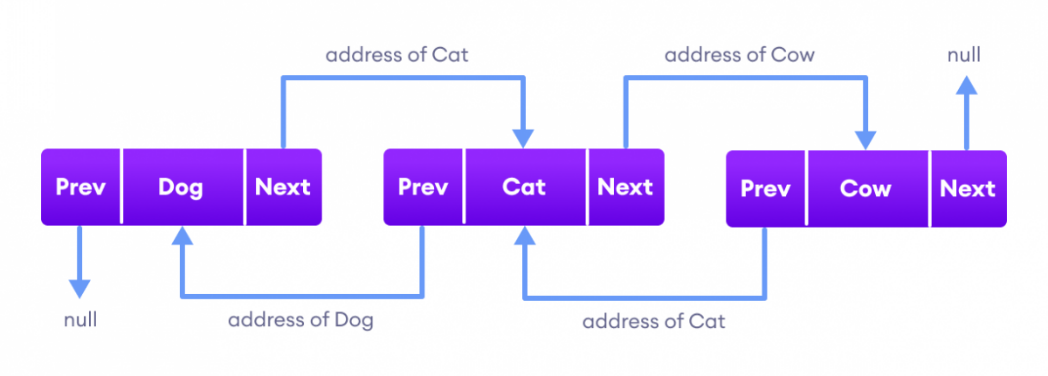
// **create Integer type linked list**

**LinkedList<Integer> linkedList = new LinkedList<>();**

**// create String type linked list**

**LinkedList<String> linkedList = new LinkedList<>();**

**Working of a Java LinkedList L:-**



Elements in linked lists are not stored in sequence. Instead, they are scattered and connected through links (Prev and Next).

**3 linkedlist nodes each connecting to one another using pointe**rs

**Here we have 3 elements in a linked list**.

**Dog** - it is the first element that holds null as previous address and the address of Cat as the next address

**Cat** - it is the second element that holds an address of Dog as the previous address and the address of Cow as the next address

**Cow** - it is the last element that holds the address of Cat as the previous address and null as the next element

* **Methods of Java LinkedList**

LinkedList provides various methods that allow us to perform different operations in linked lists. We will look at four commonly used LinkedList Operators in this tutorial:

**Add elements**

**Access elements**

**Change elements**

**Remove elements**

* **Java Vector**

The Vector class is an implementation of the [List](https://www.programiz.com/java-programming/list) interface that allows us to create resizable-arrays similar to the [ArrayList](https://www.programiz.com/java-programming/arraylist) class.

**Java Vector vs. ArrayList**

In Java, both ArrayList and Vector implements the List interface and provides the same functionalities. However, there exist some differences between them.

The Vector class synchronizes each individual operation. This means whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation.

It is because when one thread is accessing a vector, and at the same time another thread tries to access it, an exception called ConcurrentModificationException is generated. Hence, this continuous use of lock for each operation makes vectors less efficient.

However, in array lists, methods are not synchronized. Instead, it uses the Collections.synchronizedList() method that synchronizes the list as a whole.

**Note:** It is recommended to use ArrayList in place of Vector because vectors less efficient.

**Creating a Vector**

Here is how we can create vectors in Java.

**Vector<Type> vector = new Vector<>();**

**Here, Type indicates the type of a**[**linked list**](https://www.programiz.com/java-programming/linkedlist)**. For example,**

**// create Integer type linked list**

Vector<Integer> vector= new Vector<>();

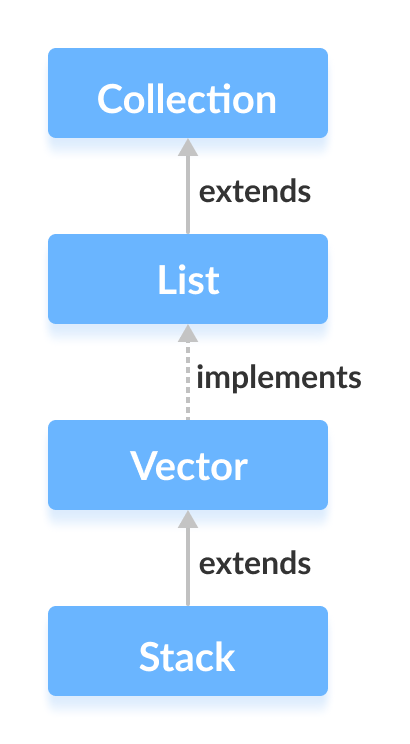
// create String type linked list

**Vector<String> vector= new Vector<>();**

* **Java Stack Class**

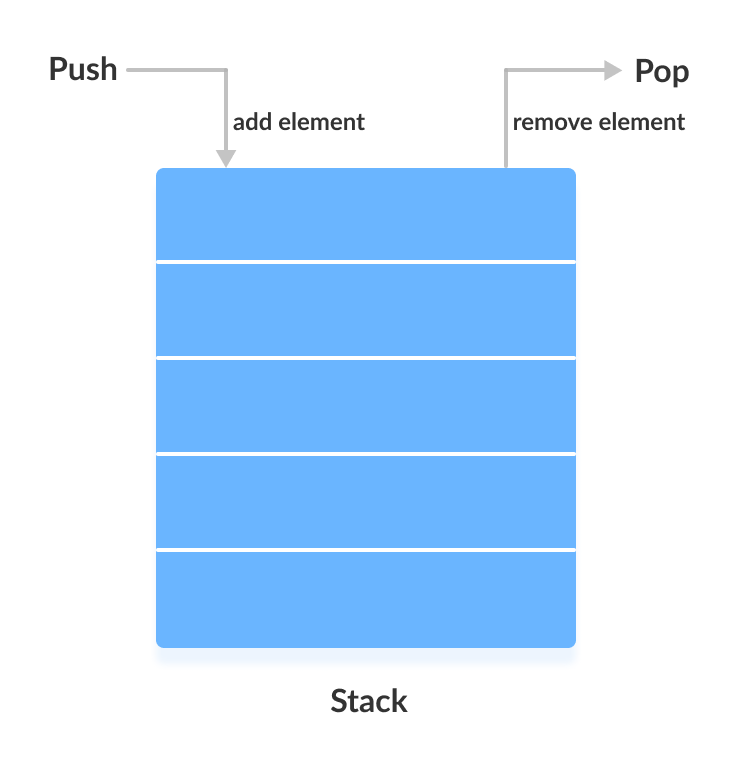
The [Java collections framework](https://www.programiz.com/java-programming/collections) has a class named Stack that provides the functionality of the stack data structure.

The Stack class extends the Vector class.



**Stack Implementation**

In stack, elements are stored and accessed in **Last In First Out** manner. That is, elements are added to the top of the stack and removed from the top of the stack.



**Creating a Stack**

In order to create a stack, we must import the java.util.Stack package first. Once we import the package, here is how we can create a stack in Java.

**Stack<Type> stacks = new Stack<>();**