**Chapter 1**

✔ What is java

✔ History of java

# Features of java

**1. Simple**

**Definition**: Java is designed to be a straightforward and accessible programming language.

**Detailed Explanation**:

* **Syntax and Structure**: Java’s syntax is derived from C and C++ but is simplified to remove complex features that often lead to errors. For instance, it eliminates explicit pointers, which can lead to memory leaks and undefined behavior.
* **Automatic Memory Management**: Java handles memory allocation and garbage collection automatically, reducing the burden on developers to manage memory manually. This reduces the risk of memory leaks and errors related to memory management.
* **No Multiple Inheritance**: Java does not support multiple inheritance (a class cannot inherit from more than one class), which simplifies the design and avoids the “diamond problem” found in languages like C++. Instead, Java uses interfaces to achieve multiple inheritance of type.

**2. Object-Oriented**

**Definition**: Java is built around the principles of object-oriented programming (OOP), which organizes software design around data, or objects, rather than functions and logic.

**Detailed Explanation**:

* **Encapsulation**: Encapsulation involves bundling the data (attributes) and methods (functions) that operate on the data into a single unit called a class. Access to the data is controlled using access modifiers like private, protected, and public, which helps in protecting the integrity of the data and exposing only what is necessary.
* **Inheritance**: Inheritance allows one class (subclass) to inherit the fields and methods of another class (superclass). This promotes code reuse and establishes a natural hierarchy. For example, a Dog class can inherit from an Animal class, gaining all its attributes and behaviors.
* **Polymorphism**: Polymorphism enables objects to be treated as instances of their parent class rather than their actual class. It allows methods to be overridden in derived classes and provides flexibility in method invocation. For example, a method makeSound() can be defined in a parent class Animal and overridden in subclasses Dog and Cat to produce different sounds.
* **Abstraction**: Abstraction involves hiding complex implementation details and showing only the essential features of an object. In Java, abstraction is achieved through abstract classes and interfaces, which define what methods a class should implement without specifying how these methods should be implemented.

**3. Platform-Independent**

**Definition**: Java applications are designed to run on any platform that has a Java Virtual Machine (JVM) without requiring modification.

**Detailed Explanation**:

* **Bytecode**: Java source code is compiled into an intermediate form called bytecode (.class files). Bytecode is platform-independent and can be executed on any system with a compatible JVM. This bytecode is portable across different operating systems and hardware configurations.
* **JVM**: The JVM is a crucial component of Java’s platform independence. It interprets or compiles bytecode into native machine code specific to the underlying hardware and operating system. By adhering to the JVM specification, Java applications can run consistently on any platform that has a JVM implementation.

**4. Secure**

**Definition**: Java incorporates various security features to protect applications from unauthorized access and malicious activities.

**Detailed Explanation**:

* **Bytecode Verification**: Before execution, Java bytecode undergoes verification to ensure it adheres to the Java language specification and does not contain illegal code that could compromise the security of the JVM.
* **Sandboxing**: Java applets and applications can run in a restricted environment called a sandbox, which limits their ability to access system resources and perform potentially harmful operations. This is especially important for applets running in a web browser.
* **Security Manager**: Java includes a security manager that allows applications to set security policies and control access to sensitive system resources like files, network connections, and system properties.

**5. Robust**

**Definition**: Java is designed to be a reliable and error-resistant programming language.

**Detailed Explanation**:

* **Exception Handling**: Java uses a structured approach to handle runtime errors through exceptions. The language provides robust exception handling mechanisms using try, catch, finally, and throw keywords, enabling developers to handle and recover from errors gracefully.
* **Automatic Garbage Collection**: Java’s garbage collector automatically reclaims memory used by objects that are no longer reachable, reducing the risk of memory leaks and ensuring efficient memory management.
* **Type Checking**: Java performs strict type checking at compile-time and runtime. This ensures that operations are performed on compatible data types and helps catch errors early in the development process.

**6. Multithreaded**

**Definition**: Java provides built-in support for concurrent execution, allowing multiple threads to run simultaneously.

**Detailed Explanation**:

* **Threading Support**: Java provides a comprehensive threading model with the java.lang.Thread class and the java.util.concurrent package. Threads allow Java applications to perform multiple tasks concurrently, improving performance and responsiveness.
* **Synchronization**: Java includes mechanisms to manage access to shared resources among threads. The synchronized keyword and concurrency utilities like ReentrantLock ensure that only one thread can access a critical section of code at a time, preventing race conditions.
* **Concurrency Utilities**: The java.util.concurrent package offers high-level concurrency utilities such as thread pools (ExecutorService), concurrent collections (ConcurrentHashMap), and atomic variables (AtomicInteger), which simplify the development of multithreaded applications.

**7. Distributed**

**Definition**: Java facilitates the development of distributed applications, where components can communicate over a network.

**Detailed Explanation**:

* **Networking Capabilities**: Java provides robust networking classes in the java.net package, including support for sockets, URLs, and HTTP. This allows Java applications to communicate over various types of networks.
* **Remote Method Invocation (RMI)**: Java RMI enables objects in one JVM to invoke methods on objects in another JVM. This facilitates the development of distributed applications where components can interact seamlessly across network boundaries.
* **Java EE (Enterprise Edition)**: Java EE (now Jakarta EE) extends Java SE with features for building enterprise-level distributed applications, such as Enterprise JavaBeans (EJBs) for business logic, JavaServer Pages (JSP) for web-based user interfaces, and Servlets for handling server-side requests.

**8. Dynamic**

**Definition**: Java supports dynamic features that allow for flexibility and adaptability in programming.

**Detailed Explanation**:

* **Dynamic Class Loading**: Java supports loading classes at runtime rather than at compile-time. This capability allows applications to dynamically load new classes and extend functionality without requiring recompilation or restarting the application.
* **Reflection**: Java provides the reflection API (java.lang.reflect) to inspect and manipulate classes, methods, and fields at runtime. Reflection is used for various purposes, including dependency injection, serialization, and creating flexible and reusable code.

**9. High Performance**

**Definition**: Java is designed to deliver high performance through various optimizations and runtime enhancements.

**Detailed Explanation**:

* **Just-In-Time (JIT) Compilation**: The JVM includes a JIT compiler that translates bytecode into native machine code at runtime. This process improves execution speed by reducing the overhead of interpreting bytecode and optimizing frequently executed code paths.
* **Optimization Techniques**: The JIT compiler and JVM perform various optimizations, such as method inlining, loop unrolling, and dead code elimination. These optimizations enhance performance by reducing execution time and resource usage.

**10. Portable**

**Definition**: Java applications are designed to be easily portable across different platforms.

**Detailed Explanation**:

* **Bytecode Portability**: Java source code is compiled into bytecode that can run on any platform with a compatible JVM. This ensures that Java applications are not tied to a specific operating system or hardware.
* **Standardization**: Java’s standard libraries and APIs provide a consistent programming interface across different platforms, ensuring that applications behave uniformly regardless of where they are executed.

**11. Interactive**

**Definition**: Java supports the development of interactive applications with user interfaces.

**Detailed Explanation**:

* **Swing**: Swing is a GUI toolkit for building rich desktop applications with a wide range of user interface components, such as buttons, text fields, tables, and trees. Swing provides a pluggable look-and-feel, allowing applications to have a consistent appearance across different platforms.
* **JavaFX**: JavaFX is a modern GUI framework that provides advanced features for building interactive and visually appealing user interfaces. It supports 2D and 3D graphics, media playback, and rich controls, making it suitable for developing sophisticated desktop and mobile applications.

1. **Architecture Neutral**

**Definition**: Java is designed to be architecture-neutral, meaning that Java programs can run on any hardware architecture without modification.

**Detailed Explanation**:

* **Bytecode Representation**: Java programs are compiled into an intermediate form called bytecode. This bytecode is platform-independent and not tied to any specific hardware architecture. It is designed to be executed by the Java Virtual Machine (JVM), which abstracts away the underlying hardware details.
* **Java Virtual Machine (JVM)**: The JVM acts as an intermediary layer between the Java bytecode and the hardware. It interprets or compiles the bytecode into native machine code suitable for the specific hardware and operating system on which the JVM is running. Because the JVM handles this translation, Java applications can run on any system with a compatible JVM, regardless of the underlying hardware architecture.
* **Portability Across Platforms**: Java’s architecture neutrality ensures that the same bytecode can be executed on different types of systems (e.g., Windows, Linux, macOS) without needing to recompile the code. This contrasts with languages that compile directly to native machine code, which require separate compilations for each hardware architecture.
* **Unified Development and Deployment**: Developers can write and compile Java programs once and deploy them across a variety of platforms without modification. This reduces development and testing overhead and promotes consistency in application behavior across different environments.

**13 Interpreted**

**Definition**: Java bytecode is interpreted or compiled to native machine code by the JVM at runtime, which means Java programs are not directly executed as native machine code but rather as bytecode that the JVM processes.

**Detailed Explanation**:

* **Interpretation Process**: When a Java program is executed, the JVM interprets the bytecode instructions one at a time. This process involves reading and executing the bytecode instructions directly, translating them into native machine instructions on-the-fly. This allows Java to be platform-independent, as the same bytecode can be interpreted differently on various hardware architectures.
* **Just-In-Time (JIT) Compilation**: To improve performance, modern JVMs use Just-In-Time (JIT) compilation. The JIT compiler translates frequently executed bytecode into native machine code during runtime. This machine code is then executed directly by the hardware, bypassing the interpretation step and improving execution speed.
* **Dynamic Adaptation**: The interpreted nature of Java allows the JVM to perform dynamic optimizations based on runtime information. The JVM can make decisions about how best to execute code based on actual usage patterns and system conditions.
* **Flexibility and Portability**: While interpretation provides flexibility and allows Java to be portable across different systems, the JIT compiler helps mitigate performance concerns by optimizing bytecode execution. The combination of interpretation and JIT compilation ensures that Java maintains both portability and performance.
* **Class Loading**: Java’s interpreted approach supports dynamic class loading, which allows classes to be loaded into the JVM as needed during program execution. This enables features like plugin architectures and dynamic extension of applications.

**13. Libraries and Frameworks**

**Definition**: Java provides a vast ecosystem of libraries and frameworks that facilitate various aspects of software development.

**Detailed Explanation**:

* **Rich API**: Java’s standard library includes a comprehensive set of APIs for various tasks, including I/O operations, networking, data manipulation, and graphical user interfaces. These APIs provide ready-to-use functionality, reducing the need for developers to implement common features from scratch.
* **Frameworks**: Java has numerous frameworks that simplify development by providing pre-built components and reducing boilerplate code. Examples include:
  + **Spring**: A framework for building enterprise applications with features like dependency injection, transaction management, and web application support.
  + **Hibernate**: An object-relational mapping (ORM) framework that simplifies database interactions by mapping Java objects to database tables.
  + **Apache Struts**: A framework for building web applications using the Model-View-Controller (MVC) pattern.

**14. Support for Modern Programming Techniques**

**Definition**: Java incorporates contemporary programming techniques to enhance productivity and code quality.

**Detailed Explanation**:

* **Functional Programming**: Introduced in Java 8, functional programming features include lambda expressions, the Stream API, and method references. These features enable developers to write concise, functional-style code and process collections in a more declarative manner.
* **Annotations**: Java supports annotations, which are metadata that provide additional information about code. Annotations can be used for various purposes, such as code generation, runtime processing, and configuration. Common examples include @Override, @Deprecated, and custom annotations used in frameworks.

**15. Community and Ecosystem**

**Definition**: Java benefits from a large, active community and a rich ecosystem that contribute to its development and support.

**Detailed Explanation**:

* **Large Community**: Java has a vast and diverse community of developers, enthusiasts, and professionals who contribute to its growth and evolution. The community provides support through forums, blogs, and open-source projects, helping developers solve problems and stay up-to-date with best practices.
* **Rich Ecosystem**: The Java ecosystem includes a wide range of tools, libraries, and frameworks that enhance development capabilities. The ecosystem supports various aspects of development, such as build tools (e.g., Maven, Gradle), IDEs (e.g., IntelliJ IDEA, Eclipse), and continuous integration tools (e.g., Jenkins).

# Usage of java

**Usage of Java**

**1. Web Development**

* **Server-Side Technologies**: Java is extensively used in web development, particularly on the server side. Frameworks like **Spring**, **JavaServer Faces (JSF)**, and **Apache Struts** help build robust, scalable web applications.
* **Servlets and JSPs**: Java Servlets and JavaServer Pages (JSP) are core technologies for building dynamic web content. Servlets handle requests and responses, while JSPs simplify the creation of HTML pages with embedded Java code.

**2. Enterprise Applications**

* **Enterprise Resource Planning (ERP)**: Java is a popular choice for building large-scale enterprise applications. It supports a range of enterprise frameworks and standards like **Java EE (Enterprise Edition)**, which includes **EJB (Enterprise JavaBeans)**, **JPA (Java Persistence API)**, and **JMS (Java Message Service)**.
* **Scalability and Reliability**: Java’s ability to handle large-scale transactions, security, and high concurrency makes it suitable for enterprise solutions.

**3. Mobile Applications**

* **Android Development**: Java has been a primary language for developing Android applications. The Android SDK provides tools and libraries for creating mobile apps on the Android platform. While Kotlin is now officially preferred, Java remains a significant part of Android development.

**4. Desktop Applications**

* **GUI Applications**: Java provides libraries like **Swing** and **JavaFX** for building cross-platform desktop applications with graphical user interfaces (GUIs). These libraries allow developers to create rich desktop applications that run on various operating systems.

**5. Scientific and Research Applications**

* **Data Analysis**: Java is used in scientific computing and data analysis due to its robustness, portability, and performance. Libraries like **Apache Commons Math** and **JAMA (Java Matrix Library)** facilitate complex calculations and data processing.
* **Simulation and Modeling**: Java's performance and extensive libraries make it suitable for simulations, modeling, and research applications.

**6. Big Data Technologies**

* **Hadoop**: Java is a key language for Hadoop, a popular framework for distributed storage and processing of large data sets. Hadoop’s components, like **MapReduce** and **HDFS (Hadoop Distributed File System)**, are written in Java.
* **Apache Kafka**: Java is also used with Apache Kafka, a distributed streaming platform that handles real-time data feeds.

**7. Embedded Systems**

* **Java ME**: Java Micro Edition (Java ME) is designed for developing applications on embedded and mobile devices. It’s used in various devices like smartcards, IoT devices, and other resource-constrained systems.

**8. Cloud Computing**

* **Cloud Platforms**: Java is commonly used for building cloud-based applications. It integrates with cloud platforms like **AWS (Amazon Web Services)**, **Google Cloud Platform**, and **Microsoft Azure** to create scalable and resilient applications.

# Java vs C++ vs Python

|  |  |  |  |
| --- | --- | --- | --- |
| Aspect | Java | C++ | Python |
| Design Philosophy | Object-oriented, platform-independent. Emphasizes code portability and security. | Multi-paradigm: object-oriented, procedural, and generic programming. Focuses on system-level programming. | High-level, interpreted, and dynamically-typed. Emphasizes readability and ease of use. |
| Syntax | C-like syntax with object-oriented constructs. | C-like syntax with added object-oriented and template features. | Clean, easy-to-read syntax with indentation for block structure. |
| Memory Management | Automatic garbage collection. | Manual memory management with pointers and explicit allocation/deallocation. | Automatic garbage collection. |
| Performance | Generally slower than C++ due to JVM overhead but faster than Python. | High performance due to direct hardware interaction and low-level system access. | Slower compared to Java and C++ due to dynamic typing and interpreter overhead. |
| Compilation | Compiled to bytecode, which runs on the Java Virtual Machine (JVM). | Compiled directly to native machine code. | Interpreted, but can be compiled to bytecode (via CPython) or machine code (via tools like Cython). |
| Execution Environment | Runs on any device with a JVM (platform-independent). | Requires compilation for specific hardware/OS (platform-dependent). | Runs on any device with a Python interpreter. |
| Object-Oriented Features | Strong support for object-oriented programming: classes, inheritance, polymorphism. | Supports object-oriented programming: classes, inheritance, polymorphism. | Supports object-oriented programming, though it's more flexible and less strict. |
| Templates/Generics | Supports generics for type-safe collections and methods. | Supports templates for generic programming and compile-time polymorphism. | No direct support for templates; uses dynamic typing and duck typing. |
| Standard Library | Extensive standard library with rich APIs for networking, I/O, and more. | Extensive standard library with powerful tools for system-level programming, but more complex. | Extensive standard library with a wide range of modules for various tasks. |
| Exception Handling | Built-in support for exception handling using try-catch blocks. | Built-in support for exception handling using try-catch blocks. | Built-in support for exception handling using try-except blocks. |
| Concurrency/Multithreading | Strong support with built-in synchronization and concurrency utilities. | Provides support for concurrency but requires manual management of threads and synchronization. | Built-in support for concurrency with threading and multiprocessing modules, though the Global Interpreter Lock (GIL) can be a limitation. |
| Memory Safety | Strong type safety, runtime checks, and automatic memory management. | No built-in safety checks; relies on programmer’s discipline. | Strong type safety, runtime checks, and automatic memory management. |
| Use Cases | Enterprise applications, web applications, Android development, large-scale systems. | System/software development, game development, performance-critical applications. | Web development, data analysis, scripting, rapid application development, automation. |
| Ease of Learning | Relatively easier to learn for beginners, especially with its object-oriented approach. | Steeper learning curve due to complex features and manual memory management. | Generally considered easier to learn due to its readable syntax and high-level nature. |
| Development Speed | Moderate speed; requires defining classes and managing object-oriented structures. | Slower development due to manual memory management and complex syntax. | Faster development due to simplicity and extensive standard libraries. |
| Cross-Platform Support | Excellent cross-platform support through JVM. | Requires recompilation for different platforms. | Good cross-platform support with Python interpreter available for most platforms. |
| Community and Ecosystem | Strong community support with a large number of frameworks and libraries. | Strong community support, especially in system-level programming and game development. | Large and active community with a vast number of third-party libraries and frameworks. |
| Integration with Other Languages | Can be integrated with native code through Java Native Interface (JNI). | Directly integrates with system and hardware; can call C and assembly code directly. | Can be integrated with C/C++ through Cython or ctypes for performance-critical sections. |
| Popular Frameworks | Spring, Hibernate, Java EE, Apache Struts. | Qt, Boost, Unreal Engine. | Django, Flask, NumPy, Pandas. |
| Development Tools | Widely supported by IDEs like Eclipse, IntelliJ IDEA, NetBeans. | Widely supported by IDEs like Visual Studio, CLion, Code::Blocks. | Widely supported by IDEs like PyCharm, VS Code, Jupyter Notebook. |

# JDK , JRE and JVM

## JDK

**1. Java Development Kit (JDK)**

The Java Development Kit (JDK) is a software development kit used to develop applications in the Java programming language. It includes a compiler, a runtime environment (Java Runtime Environment or JRE), and various tools necessary for Java application development, such as debuggers and documentation generators.

Top of Form

Bottom of Form

**Components of JDK:**

1. **Java Compiler (javac)**:
   * **Function**: Converts Java source code (.java files) into bytecode (.class files). This process is known as compilation.
   * **Output**: Bytecode files, which can be executed by the JVM.
2. **Java Virtual Machine (JVM)**:
   * **Function**: Executes the bytecode produced by the compiler. The JVM is discussed in more detail below.
   * **Included**: The JVM is part of the JRE but is also included in the JDK.
3. **Java Runtime Environment (JRE)**:
   * **Function**: Provides the libraries, Java Virtual Machine (JVM), and other components to run Java applications.
   * **Included**: The JRE is part of the JDK, and it includes the JVM along with the necessary runtime libraries.
4. **Java Standard Library**:
   * **Function**: A set of standard classes and APIs that provide basic functionalities for Java applications, such as collections, file handling, networking, and user interfaces.
   * **Components**: Includes core libraries (like java.lang, java.util, etc.) and additional libraries (like javax.swing for GUIs).
5. **Development Tools**:

|  |  |
| --- | --- |
| Tool | Description |
| appletviewer | Runs and debugs Java applets without a web browser. |
| apt | Annotation Processing Tool: Processes annotations in Java source code and generates additional code. |
| extcheck | Utility for detecting JAR file conflicts and identifying potential issues with installed extensions. |
| idlj | IDL-to-Java Compiler: Generates Java bindings from IDL (Interface Definition Language) files. |
| jabswitch | Java Access Bridge Switch: Exposes assistive technologies on Microsoft Windows systems for accessibility. |
| java | The Java Application Launcher: Executes Java applications and applets. It replaces the older jre tool, combining development and deployment functionalities. |
| javac | Java Compiler: Converts Java source code into Java bytecode, producing .class files. |
| javadoc | Documentation Generator: Automatically creates HTML documentation from Java source code comments. |
| jar | Archiver and Manager: Packages related class libraries into a single JAR (Java ARchive) file and manages JAR files. |
| javafxpackager | Packages and signs JavaFX applications for distribution. |
| jarsigner | JAR Signing and Verification Tool: Signs JAR files and verifies signatures to ensure integrity and authenticity. |
| javah | C Header and Stub Generator: Generates C header files and JNI (Java Native Interface) stubs for native methods. |
| javap | Class File Disassembler: Displays information about Java class files, including bytecode and structure. |
| javaws | Java Web Start Launcher: Launches and manages Java applications distributed via JNLP (Java Network Launch Protocol). |
| JConsole | Java Monitoring and Management Console: Monitors and manages Java applications' performance and resource usage. |
| jdb | Java Debugger: Provides a command-line interface for debugging Java programs. |
| jhat | Java Heap Analysis Tool: Analyzes heap dumps to help identify memory leaks and performance issues (experimental). |
| jinfo | Retrieves configuration information from a running Java process or a crash dump. |
| jmap | Memory Map Tool: Outputs memory map information, including details of heap memory and shared objects. |
| jmc | Java Mission Control: Monitors and analyzes Java application performance with advanced profiling capabilities. |
| jps | Java Virtual Machine Process Status Tool: Lists HotSpot JVMs on the system and provides process details. |
| jrunscript | Command-Line Script Shell: Allows execution of JavaScript and other scripting languages from the command line. |
| jstack | Prints Java stack traces of Java threads, useful for diagnosing thread-related issues (experimental). |
| jstat | JVM Statistics Monitoring Tool: Monitors various JVM statistics, such as garbage collection and memory usage (experimental). |
| jstatd | jstat Daemon: Provides a network interface to monitor JVM statistics (experimental). |
| keytool | Keystore Manipulation Tool: Manages keystore files and certificates used for cryptographic operations. |
| pack200 | JAR Compression Tool: Compresses JAR files to reduce their size. |
| Policytool | Policy Creation and Management Tool: Defines and manages security policies for Java runtime environments. |
| VisualVM | Visual Tool: Integrates multiple JDK tools and provides profiling and monitoring of Java applications. |
| wsimport | Generates JAX-WS (Java API for XML Web Services) artifacts for web service invocation and communication. |
| xjc | JAXB (Java Architecture for XML Binding) Tool: Converts XML schema into Java classes for XML processing. |

**Usage:**

* **Development**: Used by developers to create and compile Java applications.
* **Installation**: Requires installation on the developer’s machine and is not required for end-users who only need to run Java applications.

## JRE

The Java Runtime Environment (JRE) is a software platform that enables the execution of Java applications. It works by providing a runtime environment that includes the Java Virtual Machine (JVM) and a set of core libraries. The JRE allows Java programs to run on any device or operating system without modification, by translating Java bytecode into native machine code that the JVM executes.

* Technologies which get used for deployment such as Java Web Start.
* Toolkits for user interface like Java 2D.
* Integration libraries like **Java Database Connectivity (JDBC)** and **Java Naming and Directory Interface (JNDI)**.
* Libraries such as Lang and util.
* Other base libraries like **Java Management Extensions (JMX)**, **Java Native Interface (JNI) and Java for XML Processing (JAX-WS)**.

**Components of JRE**

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

**Definition:** The JVM is the core component of the JRE that provides a runtime environment for executing Java bytecode. It abstracts the underlying hardware and operating system, allowing Java applications to run on any platform that has a compatible JVM.

**Functionality:**

* **Bytecode Execution:** The JVM interprets or compiles Java bytecode into machine code that the host operating system can execute.
* **Memory Management:** It handles memory allocation and garbage collection to manage resources efficiently.
* **Platform Independence:** Ensures that Java applications can run on any device or operating system with a suitable JVM, adhering to the "write once, run anywhere" principle.

**2. Java Class Libraries**

**Definition:** These are a comprehensive set of pre-written classes and APIs (Application Programming Interfaces) that provide a wide range of functionalities for Java applications. They are bundled with the JRE and are essential for performing common tasks.

**Functionality:**

* **Core Libraries:** Include foundational classes for basic tasks such as data types, collections, file I/O, networking, and string manipulation.
* **Extended Libraries:** Provide additional functionalities, such as graphical user interface components (Swing, JavaFX), database access (JDBC), and security features (Java Cryptography Architecture).

**3. Java Class Loader**

**Definition:** The Java Class Loader is a part of the JVM responsible for dynamically loading Java classes into memory at runtime.

**Functionality:**

* **Loading Classes:** It loads class files from the local filesystem, network, or other sources when they are needed by the application.
* **Verification:** Ensures that the loaded classes adhere to the Java language specification and do not violate security constraints.
* **Initialization:** Initializes classes and their static variables before they are used by the application.

**4. Java Runtime Components**

**Definition:** These are the essential runtime components and tools that support the execution of Java applications, including configuration files and native libraries.

**Functionality:**

* **Configuration Files:** Contain settings and preferences that configure the JVM and runtime behavior, such as the java.security file for security policies and the logging.properties file for logging configuration.
* **Native Libraries:** Platform-specific libraries written in native languages (e.g., C or C++) that the JVM uses to interact with the underlying operating system and hardware.
* **Support Utilities:** Include tools and utilities that help manage and configure the runtime environment, such as the java command for launching applications and the java command-line tool for managing the JRE.

## Jvm

JVM, or Java Virtual Machine, is an essential component of the Java Runtime Environment (JRE). It's a virtual machine that provides an execution environment for Java bytecode, the compiled form of Java source code. Here's why JVM is necessary and helpful in Java development:

1. **Platform Independence**: One of the key advantages of Java is its platform independence. Java source code is compiled into bytecode, which is platform-neutral. The JVM then executes this bytecode on any platform that has a compatible JVM implementation. This "write once, run anywhere" capability allows Java programs to run on diverse hardware and operating systems without modification.
2. **Memory Management**: The JVM manages memory allocation and deallocation dynamically, making memory management transparent to the developer. It handles tasks such as allocating memory for objects, garbage collection (reclaiming memory occupied by objects no longer in use), and optimizing memory usage to minimize overhead and improve performance.
3. **Security**: The JVM provides built-in security features to protect against malicious code execution. It enforces security policies, such as access control and code verification, to prevent unauthorized access to system resources and protect against potential security vulnerabilities.
4. **Optimization and Performance**: The JVM includes sophisticated runtime optimizations to improve the performance of Java applications. It employs techniques such as just-in-time (JIT) compilation, which dynamically compiles bytecode into native machine code for execution, and adaptive optimization, which adjusts optimization strategies based on runtime profiling data. These optimizations help Java programs achieve competitive performance comparable to natively compiled languages.
5. **Portability**: By abstracting the underlying hardware and operating system details, the JVM ensures that Java applications behave consistently across different platforms. Developers can write Java code once and deploy it on any JVM-compatible platform without worrying about platform-specific dependencies or differences.
6. **Dynamic Loading and Linking**: The JVM supports dynamic class loading and linking, allowing Java applications to load classes and resources dynamically at runtime. This enables features such as dynamic extension, plugin systems, and modular development, where components can be added or removed without restarting the application.
7. **Debugging and Profiling**: The JVM provides debugging and profiling capabilities to aid developers in diagnosing and optimizing Java applications. Debugging tools allow developers to inspect variables, set breakpoints, and trace program execution, while profiling tools provide insights into resource usage, performance bottlenecks, and memory consumption.

**When JVM starts it performs some specific tasks**

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

**JVM provides definitions for the**

* Memory area
* Class file format
* Register set
* Garbage-collected heap
* Fatal error reporting etc.

## **Architecture of JVM**

A diagram of a computer system

Description automatically generated

**Some short points about JVM**

1. It’s a runtime engine responsible to run java-based applications
2. Main tasks of **jvm** is

* Load .class file
* Execute .class file

1. Virtual machine is a software simulation of a machine which can perform operations

Like physical machine.

1. Virtual machine is not physically present.
2. A virtual machine, usually known as a guest is created within another computing.
3. Environment referred as a "**host**." multiple virtual machines can exist within a single host at one time.

* For e.g. Calculator software in operating system, which is not physically present, but performs all the functions like physical calculator.

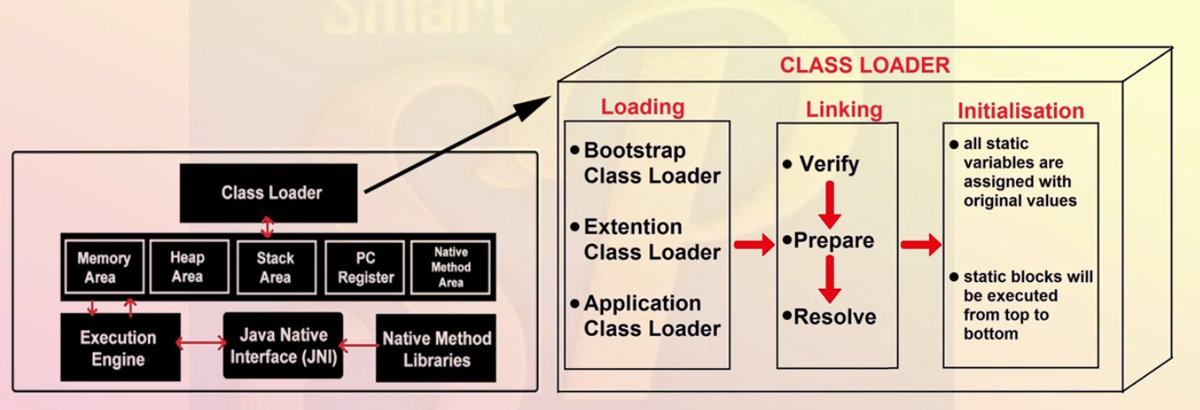
1. Types of virtual machine:

* Hardware based or system based virtual machine
* Application based or process based virtual machine

1. JVM (java virtual machine) is application based virtual machine.

Let’s start from **Class Loader**

### **Class Loader**



Class Loader It is responsible for the following three tasks

1. Loading
2. Linking
3. Initialisation
4. **Loading**

• It will read .class file and store corresponding information in the method area.

• For each class file, JVM will store following information in the method area.

1. *Fully qualified class name*
2. *Fully qualified parent class name*
3. *Methods information*
4. *Variables information*
5. *Constructors’ information*
6. *Modifiers information*
7. *Constant pool information*

• Three types of class loaders

1. **Bootstrap Class Loader**

Responsible to load the classes present in rt.jar ( **rt.jar** file is present in bootstrap classpath i.e**. jdk\jre\lib** )

1. **Extention Class Loader**

Responsible to load the classes from extension class path ( i.e. **jdk\jre\lib\ext\\*.jar**)

1. **Application Class Loader**

Responsible to load the classes from application class path. It internally uses environment variable class path.

1. **Linking**

• In linking three activities are performed

1. *Verification*
2. *Preparation*
3. *Resolution*

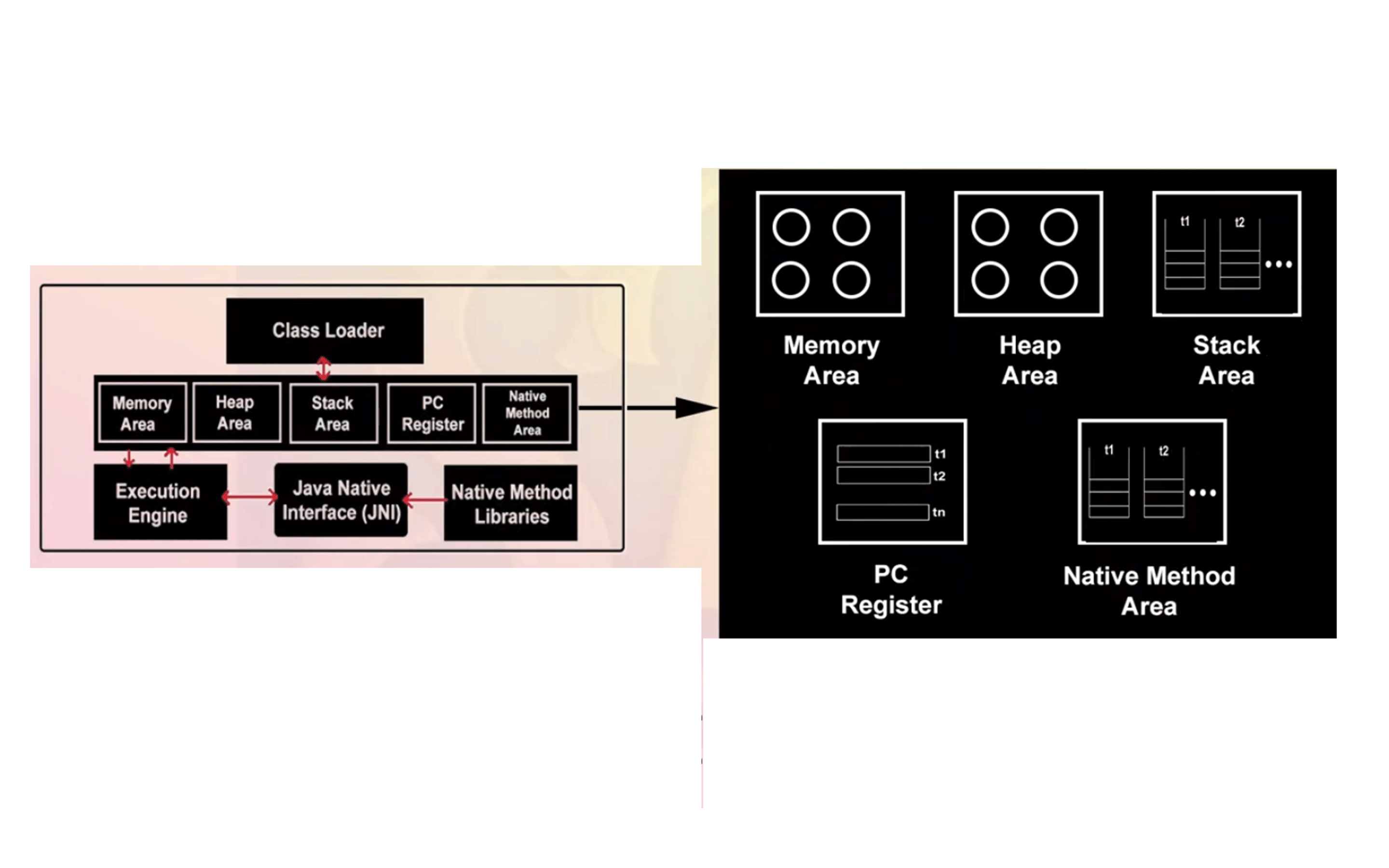
* **Verification** In this process Byte Code Verifier Checks Whether the -class file is generated by valid compiler or not and whether \*Class file is properly formatted or not. If verification fails, then JVM will provide "**java.Iang.VerifyError**" exception. Because of this process, java is secured.
* **Preparation** In this process JVM will allocate memory for class level static variables & assign default values (not original values).
* **Resolution** In this process symbolic names present in our program are replaced with original memory references from method area.

1. **Initialisation**

In this process, two activities will be performed

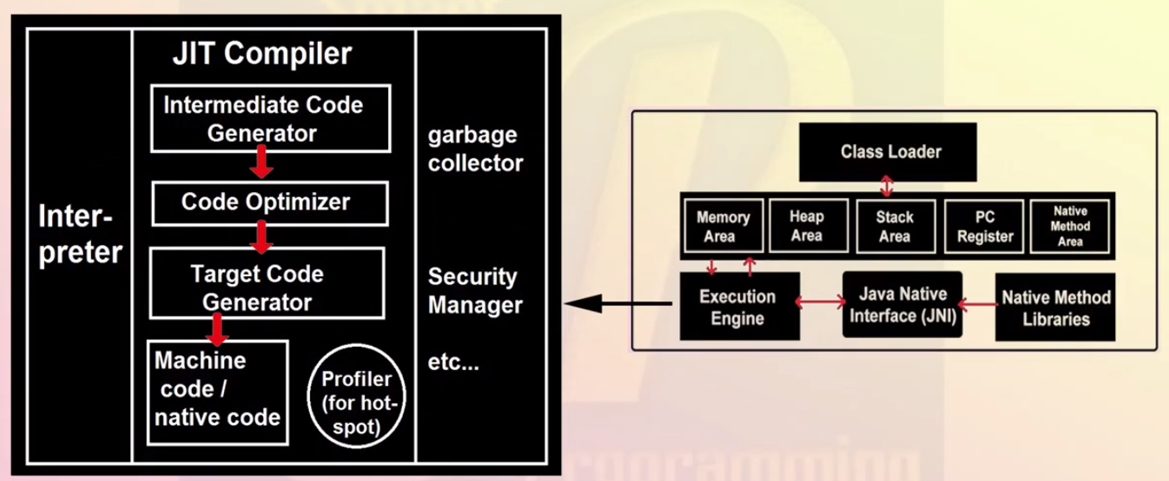
1. All static variables are assigned with original values.
2. static blocks will be executed from top to bottom

### **Memory Areas**



1. **Method Area**
2. Method area is created when JVM is started.
3. It stores **.class** file information and static variables.
4. Per JVM one memory area, therefore multiple threads can access this area, so it is not thread safe.
5. **Heap Area**
6. Heap area is created when JVM is started.
7. It stores objects, instance variables and arrays (as every array is an object in java).
8. It can be accessed by multiple threads, so the data stored in heap area is not thread safe.
9. **Stack Area**
10. Whenever a new thread is created, a separate stack area will also be created It stores the current running method and local variables.
11. When the method is completed, the corresponding entry from the stack will be removed After completing all method calls, the stack will become empty, and that empty stack will be destroyed by thee JVM just before terminating the thread.
12. The data stored in the stack is available only for the corresponding thread and not available to the remaining thread, so this area is thread safe.
13. **PC (Program Counter) Register**
14. It holds the address of next executing instruction for every thread, a separate pc register is created, so it is also thread safe.
15. **Native Method Stacks**
16. All native method calls invoked by the thread will be stored in the corresponding native method stack.
17. For every thread separate native method stack will be created It is also thread safe.

### **Execution Engine**



* Execution Engine is responsible to execute java class file.
* It contains mainly two components

1. Interpreter
2. JIT Compiler
3. **Interpreter**

* A module that alternately decodes and executes every statement or line in somebody of code. The Java interpreter decodes and executes bytecode for the Java virtual machine.



1. **JIT Complier**

* JIT stands for Just-in-Time which means that code gets compiled when it is needed, not before runtime.
* The main purpose of JIT compiler is to improve performance.
* JVM maintains a count as of how many times a function is executed. If this count exceeds a predefined limit or say threshold value, the JIT compiles the code into machine language which can directly be executed by the processor (unlike the normal case in which **javac** compile the code into bytecode and then java - the interpreter interprets this bytecode line by line converts it into machine code and executes).
* Also, next time this function is calculated same compiled code is executed again unlike normal interpretation in which the code is interpreted again line by line. This makes execution faster.
* JIT compilation is applicable only for repeatedly required methods, not for every method.

### **JNI Java Native Interface**

* An interface that allows Java to interact with code written in another language.
* It acts as mediator for java method calls & the corresponding native libraries i.e. JNI is responsible to provide information about native libraries to the JVM.
* Native Method Library provides or holds native library information.
* The java command-line utility is an example of one such application, that launches Java code in a Java Virtual Machine.

# How java main method works

The Java main function is a critical component for executing Java programs. It serves as the entry point and is fundamental for understanding how Java applications are started. This comprehensive guide will explore each keyword used in the main function, including the class keyword which is crucial for context.

**I. The Java main Method: An Overview**

The main method is the entry point for any standalone Java application. It serves as the starting point where the Java Virtual Machine (JVM) begins the execution of a program. To appreciate its importance, let’s dissect its structure and each of its components.

**A. Structure of the main Method**

Here is the typical structure of the main method:

public class HelloWorld {

public static void main(String[] args) {

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

}

}

In this example, we have a class named HelloWorld, and within this class, there is a main method. The method prints "Hello, World!" to the console.

**B. Breaking Down the main Method**

**1. public Keyword**

The public keyword is an access modifier that makes the main method accessible from any other class. In Java, access modifiers determine the visibility of classes, methods, and other members. By declaring the main method as public, we ensure that the JVM can invoke it from outside the class.

**Why Use public?**

* **Visibility:** It allows the JVM to access the main method to start program execution.
* **Conventional Practice:** Adheres to Java's requirement for the main method to be accessible.

**2. static Keyword**

The static keyword denotes that the main method is a class method, not an instance method. This means it can be called directly by the JVM without creating an instance of the class.

**Why Use static?**

* **Class-Level Method:** The method can be executed without needing an instance, which is crucial since the JVM does not instantiate the class before calling main.
* **Efficiency:** Eliminates the overhead of object creation just to execute the method.

**3. void Keyword**

The void keyword specifies that the main method does not return any value. It performs operations, such as executing code or printing output, without producing a return result.

**Why Use void?**

* **Purpose:** The main method’s purpose is to run the program, not to return a value.
* **Consistency:** Follows Java's convention for the main method, which does not return a value.

**4. main Method Name**

The name main is a special identifier recognized by the JVM as the entry point of the program. It must be exactly main to conform to Java standards.

**Why main?**

* **Standard Entry Point:** The JVM looks for this specific name to start the application.
* **Conformity:** Ensures the method is properly recognized as the starting point.

**5. String[] args Parameter**

The String[] args parameter is an array of String objects used to pass command-line arguments to the program. This allows users to provide input data when the program starts.

**Why Use String[] args?**

* **Command-Line Input:** Facilitates dynamic input to customize program behavior.
* **Flexibility:** Enables the program to handle various scenarios based on user input.

**II. The Role of the class Keyword**

The class keyword is fundamental in Java as it defines a blueprint for creating objects. It encapsulates data and methods, including the main method, providing structure and organization.

**A. Structure of a Java Class**

A Java class encapsulates data and methods. The main method is typically enclosed within a class. Here’s a breakdown of the structure:

public class HelloWorld {

public static void main(String[] args) {

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

}

}

In this example, HelloWorld is a class that contains the main method.

**B. Breaking Down the class Keyword**

**1. Definition**

The class keyword defines a new class. A class serves as a template from which objects are created.

**Why Use class?**

* **Encapsulation:** Groups data and methods together, improving code organization.
* **Object-Oriented Design:** Supports the principles of encapsulation, inheritance, and polymorphism.

**2. Access Modifiers for Classes**

Classes, like methods, can be defined with access modifiers such as public, protected, and private. In our example, public ensures that the class can be accessed from other classes.

**Why Use Access Modifiers?**

* **Visibility:** Controls the access level of the class, ensuring it is accessible where needed.
* **Encapsulation:** Enhances the design by controlling how classes interact with each other.

**III. How the JVM Utilizes the main Method**

Understanding how the JVM interacts with the main method is crucial for grasping Java program execution.

**A. Program Execution Flow**

When you run a Java application, the JVM performs the following steps:

1. **Class Loading:** The JVM loads the class containing the main method.
2. **Method Invocation:** The JVM looks for the public static void main(String[] args) method.
3. **Execution:** The JVM executes the code inside the main method.

**B. Importance of the main Method**

The main method serves as the entry point, making it essential for standalone Java applications. Without it, the JVM would not know where to start executing the program

**IV. Practical Examples and Use Cases**

Let’s look at a few practical examples to understand how the main method and class keyword are used in real-world applications.

**A. Basic Example**

Consider a simple program that prints "Hello, World!" to the console:

public class HelloWorld {

public static void main(String[] args) {

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

}

}

**B. Command-Line Arguments Example**

Here’s an example that demonstrates how to use command-line arguments:

public class Greeting {

public static void main(String[] args) {

if (args.length > 0) {

System.out.println("Hello, " + args[0] + "!");

} else {

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

}

}

}

In this example, the program greets the user with a personalized message if a command-line argument is provided.

# How to set path of java ?

Setting the Java PATH environment variable on a Windows system is a crucial step to ensure that Java commands like javac and java can be executed from any Command Prompt window. Here’s a detailed, step-by-step guide on how to set the PATH for Java on Windows:

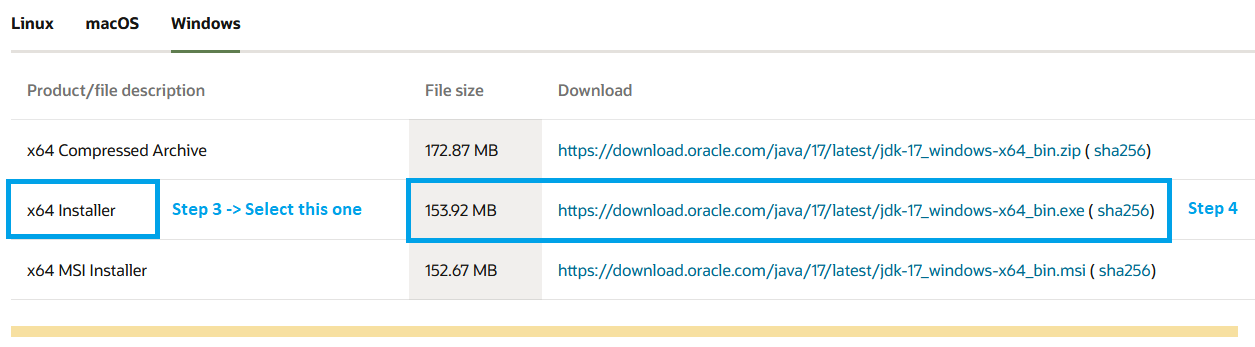
**1. Install Java Development Kit (JDK)**

1. **Download JDK**: Go to the [Oracle JDK download page](https://www.oracle.com/java/technologies/javase-downloads.html) or the [OpenJDK website](https://openjdk.java.net/) and download the installer for your version of Windows.
   1. **Step 1**

A screenshot of a computer

Description automatically generated

**Step 2**

****

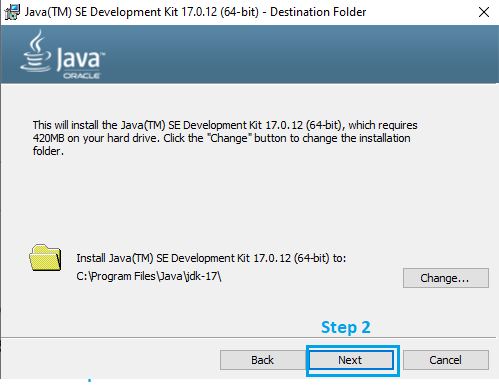
1. **Run the Installer**: Execute the downloaded installer and follow the installation instructions. Note the installation directory, typically something like C:\Program Files\Java\jdk-<version>.

**Step 1**

A screenshot of a computer

Description automatically generated

**Step 2**

****

**Step 3**

Now your program is installing

A screenshot of a software update

Description automatically generated

**Step 4**

****

**2. Set JAVA\_HOME Environment Variable**

1. **Open System Properties**:
   * Press Win + R to open the Run dialog.

A screenshot of a computer error

Description automatically generated

* + Type sysdm.cpl and press Enter. This opens the System Properties window.

A screenshot of a computer error

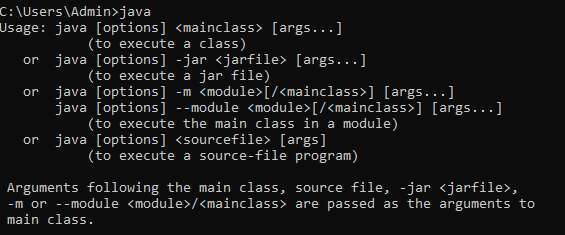
Description automatically generated

1. **Access Environment Variables**:
   * In the System Properties window, go to the **Advanced** tab.
   * Click on **Environment Variables**.
2. **Create JAVA\_HOME**:
   * In the Environment Variables window, click **New** under the **System variables** section.
   * Set the **Variable name** to JAVA\_HOME.
   * Set the **Variable value** to the path of your JDK installation, e.g., C:\Program Files\Java\jdk-<version>.
   * Click **OK** to save.

**3. Update the PATH Variable**

1. **Locate PATH Variable**:
   * In the Environment Variables window, find the Path variable in the **System variables** section and select it.
   * Click **Edit**.
2. **Add Java Bin Directory to PATH**:
   * In the Edit Environment Variable window, click **New**.
   * Add the path to the JDK bin directory. This is typically C:\Program Files\Java\jdk-<version>\bin.
   * Click **OK** to save.

**4. Verify the Configuration**

1. **Open Command Prompt**:
   * Press Win + R to open the Run dialog.
   * Type cmd and press Enter to open the Command Prompt.
2. **Check JAVA\_HOME**:
   * Type java and press Enter. This should display the java information.
   * 
   * If you want to check java version then type java -version in command prompt to check the version.
   * A screen shot of a computer

     Description automatically generated
3. **Check PATH**:
   * Type java -version and press Enter. You should see the installed version of Java displayed, confirming that Java is correctly set up in your PATH.
   * Type javac -version and press Enter. You should also see the version of the Java compiler.

**Troubleshooting Tips**

* **Reboot if Necessary**: Sometimes, a reboot might be required for changes to take effect.
* **Ensure No Typographical Errors**: Double-check the paths and variable names for any typographical errors.
* **Administrator Privileges**: Ensure you have administrator privileges when setting system-wide environment variables.

# How to compile and run java program

**Prerequisites**

* Ensure that the Java Development Kit (JDK) is installed on your system. The JDK includes the Java compiler (javac) and the Java runtime (java).
* Make sure the JDK's bin directory is added to your system's PATH environment variable. This allows you to run javac and java commands from any directory.

**Steps to Compile and Run a Java Program**

1. **Write Your Java Program**
   * Open a text editor (such as Notepad) and write your Java code.
   * Save the file with a .java extension, for example, HelloWorld.java.

**Example Code (HelloWorld.java):**

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

1. **Open Command Prompt**
   * Press Win + R, type cmd, and press Enter to open the Command Prompt.
2. **Navigate to the Directory Containing Your Java File**
   * Use the cd command to change directories to where your .java file is located. For example:

cd path\to\your\java\file

1. **Compile the Java Program**
   * Use the javac command to compile the .java file. This command generates a .class file with bytecode that the JVM can execute.

javac HelloWorld.java

* + If there are no errors in your code, this command will create a HelloWorld.class file in the same directory.

1. **Run the Java Program**
   * Use the java command to run the compiled Java program. Do not include the .class extension.

java HelloWorld

* + The output should be:

Hello, World!

**Troubleshooting**

* **Command Not Found:** If you receive an error saying 'javac' is not recognized as an internal or external command, check that the JDK bin directory is correctly added to your PATH environment variable.
* **Compilation Errors:** If there are syntax errors or issues in your code, the javac command will display error messages. Correct these errors in your .java file and try compiling again.
* **Class Not Found:** If you receive an error saying Could not find or load main class, ensure that the class name matches the filename (e.g., HelloWorld in HelloWorld.java), and that you’re running the java command from the directory containing the .class file.

# How java works

Understanding how Java works involves exploring its architecture and the processes involved in running a Java application. Java is known for its "write once, run anywhere" philosophy, which is achieved through a combination of compilation and interpretation using the Java Virtual Machine (JVM). Here’s a detailed explanation of how Java works:

**1. Java Source Code**

Java programs start as plain text files with a .java extension. These files contain Java source code written in a human-readable form. This source code defines classes and methods that form the basis of a Java application.

**Example of a Java Source Code File (HelloWorld.java):**

Understanding how Java works involves exploring its architecture and the processes involved in running a Java application. Java is known for its "write once, run anywhere" philosophy, which is achieved through a combination of compilation and interpretation using the Java Virtual Machine (JVM). Here’s a detailed explanation of how Java works:

**1. Java Source Code**

Java programs start as plain text files with a .java extension. These files contain Java source code written in a human-readable form. This source code defines classes and methods that form the basis of a Java application.

**Example of a Java Source Code File (HelloWorld.java):**

public class HelloWorld {

public static void main(String[] args) {

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

}

}

**2. Compilation**

Java source code is compiled into bytecode using the Java compiler (javac), which is part of the JDK (Java Development Kit).

* **Compilation Process:**
  1. **Source Code Compilation**: The Java compiler (javac) translates the .java file into bytecode, which is stored in a .class file. Bytecode is an intermediate, platform-independent code that the JVM can understand.
  2. **Bytecode**: This .class file contains bytecode instructions that are not directly executable by the hardware but are understood by the JVM.

**Example Compilation Command:**

javac HelloWorld.java

This command generates a HelloWorld.class file.

**3. Java Virtual Machine (JVM)**

The JVM is an abstract computing machine that enables a computer to run Java programs. The JVM interprets and executes Java bytecode.

* **JVM Functions:**
  1. **Loading**: The JVM loads .class files (bytecode) into memory. The class loader is responsible for locating and loading the classes.
  2. **Verification**: The bytecode verifier checks the code to ensure it adheres to Java’s security and runtime constraints. This helps prevent malicious code from compromising the JVM or the host system.
  3. **Execution**: The JVM interprets the bytecode or uses Just-In-Time (JIT) compilation to convert bytecode into native machine code for execution. This allows Java programs to run on any platform with a compatible JVM.

**4. Just-In-Time (JIT) Compilation**

JIT compilation is a performance optimization technique used by the JVM to improve the runtime performance of Java applications.

* **How JIT Works:**
  1. **Hot Code Paths**: The JVM identifies frequently executed parts of the code (hot spots) and compiles them into native machine code.
  2. **Native Code Execution**: The compiled native code is executed directly by the operating system, which is faster than interpreting bytecode.

**5. Execution Process**

Here’s a step-by-step overview of what happens when a Java program is executed:

1. **Start JVM**: When you run a Java program, the JVM is started. It sets up an execution environment.
2. **Class Loading**: The JVM’s class loader loads the .class files. It starts with the main class specified by the user.
3. **Bytecode Verification**: The JVM verifies the bytecode to ensure it adheres to Java’s security and integrity rules.
4. **Execution**: The JVM either interprets the bytecode or uses JIT compilation to convert it into native code and execute it.
5. **Garbage Collection**: The JVM automatically manages memory by performing garbage collection, which reclaims memory used by objects no longer in use.

**6. Java Runtime Environment (JRE) and Java Development Kit (JDK)**

* **JRE**: Includes the JVM and libraries necessary to run Java applications. It does not include development tools like compilers and debuggers.
* **JDK**: Includes the JRE plus development tools such as the Java compiler (javac), debugger (jdb), and other tools for developing Java applications.

**7. Platform Independence**

Java achieves platform independence through the use of bytecode and the JVM. The same .class file (bytecode) can be run on any platform with a compatible JVM. This is achieved by:

* **Compilation to Bytecode**: Java source code is compiled into platform-independent bytecode.
* **Execution on JVM**: The JVM on any platform interprets or compiles this bytecode to native code suitable for the specific operating system and hardware.

**8. Memory Management**

Java manages memory automatically through garbage collection. The JVM allocates memory for objects and reclaims memory occupied by objects that are no longer in use, which helps prevent memory leaks and reduces manual memory management tasks for developers.

# Variables

Variables in Java are fundamental components that store data values. They act as containers that hold information that can be referenced and manipulated in a program. Understanding how variables work is key to writing effective Java code.

1. In Java variable is a name given to a memory location.
2. It is the basic unit of storage in a program.
3. The value stored in a variable can be changed during program execution.
4. Variables in Java are only a name given to a memory location.
5. All the operations done on the variable affect that memory location.
6. In Java, all variables must be declared before use.

Key Concepts of Variables in Java:

1. **Variable Declaration:**

* A variable must be declared with a specific data type before it can be used.
* The declaration specifies the type of data the variable will hold (e.g., `**int**`, `**String**`, `**double**`,**’Boolean’**, **’char’**).
  + int number; // Declares a variable 'number' of type int

1. **Variable Initialization:**

* A variable can be initialized at the time of declaration or later in the code.
* Initialization assigns a value to the variable.
  + int number = 10; // Declares and initializes the variable 'number' with the value 10

1. **Types of Variables:**

Java variables are categorized based on their scope and lifetime. Mainly there are three types of variables are available in java :

1. **Instance Variables (Non-Static Fields)**

* **Definition:** Fields declared without the static keyword.
* **Belong to Objects:** Each object of a class has its own copy of instance variables.
* **Unique Values:** The value of an instance variable can vary from one object to another.
* **Default Values:** If not explicitly initialized, instance variables are automatically assigned default values based on their data type (e.g., 0 for int, null for objects).
* **Access:** Instance variables can be accessed by methods within the same class, typically using the this keyword.
* **Lifecycle:** They are created when an object is instantiated and destroyed when the object is garbage collected.
* **Visibility:** Can be controlled using access modifiers (e.g., private, protected, public).
* **Example:**

class Bicycle {

int currentSpeed; // Instance variable, unique to each Bicycle object

}

1. **Class Variables (Static Fields):**

* **Definition:** Fields declared with the static keyword, meaning they belong to the class rather than any object.
* **Single Copy:** There is only one copy of a static variable, regardless of how many instances of the class are created.
* **Shared Among Instances:** All objects of the class share the same value of the static variable.
* **Access:** Static variables can be accessed directly by the class name (e.g., ClassName.staticVariable) and also by objects, although the class name is preferred.
* **Default Values:** Automatically assigned default values if not explicitly initialized, similar to instance variables.
* **Memory Allocation:** Memory for static variables is allocated when the class is loaded by the JVM, and they exist as long as the class is loaded.
* **Use Cases:** Commonly used for constants or shared properties, such as configuration settings or counters.
* **Example:**

class Bicycle {

static int numGears = 6; // Class variable, shared by all Bicycle objects

}

1. **Local Variables:**

* **Definition:** Variables declared within a method, constructor, or block of code.
* **Scope:** Only accessible within the method, constructor, or block where they are declared.
* **No Default Values:** Local variables must be explicitly initialized before use, as they do not have default values.
* **Lifecycle:** Created when the method, constructor, or block is entered and destroyed when it exits.
* **No Access Modifiers:** Cannot have access modifiers because their scope is limited to the method or block they are declared in.
* **Memory Allocation:** Memory for local variables is allocated on the stack when the method is called and deallocated when the method finishes execution.
* **Efficiency:** Local variables are often more efficient than instance or static variables due to their limited scope and lifespan.
* **Example:**

1. void changeGear(int newGear) {

2. int gear = newGear; // Local variable, only accessible within this method

3. }

4.

1. **Final Variables (Special Case for Constants):**

* **Definition:** Variables declared with the final keyword, indicating their value cannot be changed once assigned.
* **Constant Value:** Often used for constants where the value is set once and should not change throughout the program.
* **Instance Final Variables:** Each object can have its own final instance variable, which can be initialized either during declaration or in the constructor.
* **Static Final Variables:** When combined with static, the final variable becomes a constant shared across all instances of the class.
* **Naming Convention:** Typically named using all uppercase letters with words separated by underscores (e.g., MAX\_SPEED).
* **Example:**

1. class Bicycle {

2. static final int MAX\_SPEED = 100;

// Static final variable, a constant shared by all Bicycle objects

3. }

4.

1. **Variable naming conventions**
2. **Case Sensitivity:**

* Variable names in Java are case-sensitive.

1. **Legal Identifiers:**

* Variable names can be any legal identifier, which is an unlimited-length sequence of Unicode letters and digits.
* The name must begin with a letter, the dollar sign ($), or the underscore character (\_).

1. **Naming Conventions:**

* It is recommended to start variable names with a letter, not with $ or \_.
* The $ character is generally avoided in variable names, except in auto-generated names.
* Although \_ is allowed, starting variable names with \_ is discouraged.
* White space is not permitted in variable names.

1. **Subsequent Characters:**

* After the first character, variable names can include letters, digits, dollar signs ($), or underscore characters (\_).

1. **Readable Names:**

* Use full words in variable names instead of cryptic abbreviations to make the code more readable and understandable.
* This practice helps in making the code self-documenting.

1. **Avoid Keywords:**

* Variable names must not be keywords or reserved words in Java.

1. **Single Word Naming:**

* If the variable name consists of only one word, it should be in all lowercase letters.

1. **Multiple Word Naming:**

* If the variable name consists of more than one word, capitalize the first letter of each subsequent word (e.g., gearRatio, currentGear).

1. **Constant Variables:**

* For constants (static final), use all uppercase letters and separate words with an underscore (\_) (e.g., NUM\_GEARS).
* The underscore character is typically used only in constants, not elsewhere in variable names.

1. **Scope of Variables:**

* The scope of a variable is the region in the code where the variable is accessible.
* Local variables have a limited scope (within a method or block).
* Instance variables are accessible throughout the class (unless restricted by access modifiers).
* Static variables are accessible across all instances and can be accessed without creating an object of the class.

1. **Example of Variables in Java:**

public class Example {

static int staticVar = 20; // Static variable

int instanceVar = 10; // Instance variable

public void display() {

int localVar = 5; // Local variable

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

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

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

}

public static void main(String[] args) {

Example obj = new Example();

obj.display();

}

}

**Output:**

Local Variable: 5

Instance Variable: 10

Static Variable: 20

# Typecasting in java

**What is Type Casting in Java?**

Type casting in Java is the process of converting one data type into another. It's a way to convert a variable from one type to another type. This is often necessary when you need to perform operations between different data types or when you need to assign a value of one type to a variable of another type.

**Types of Type Casting in Java**

There are two main types of type casting in Java:

1. **Widening (Automatic) Type Casting**
2. **Narrowing (Explicit) Type Casting**

**1. Widening (Automatic) Type Casting**

Widening type casting happens automatically when a smaller data type is assigned to a larger data type. Since there is no risk of data loss, Java automatically handles the conversion.

* **Description:** Smaller data types are automatically converted into larger data types.
* **No Data Loss:** There is no loss of information because the target data type can accommodate the value of the source data type.
* **Automatic Conversion:** The conversion happens automatically without the need for explicit code.

**Example of Widening Casting:**

* byte to short, int, long, float, or double
* short to int, long, float, or double
* char to int, long, float, or double
* int to long, float, or double
* long to float or double
* float to double

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

int intVal = 100;

long longVal = intVal; // int to long float

floatVal = longVal; // long to float

double doubleVal = floatVal; // float to double

System.out.println("int value: " + intVal);

System.out.println("long value: " + longVal);

System.out.println("float value: " + floatVal);

System.out.println("double value: " + doubleVal);

**2. Narrowing (Explicit) Type Casting**

Narrowing type casting needs to be done manually by the programmer. This type of casting involves converting a larger data type into a smaller data type. Since there is a risk of data loss, the conversion must be explicitly stated in the code.

* **Description:** Larger data types are manually converted into smaller data types.
* **Possible Data Loss:** There can be data loss or overflow if the value of the larger data type does not fit into the smaller data type.
* **Explicit Conversion:** The conversion requires explicit casting syntax.

**Example of Narrowing Casting:**

* short to byte or char
* char to byte or short
* int to byte, short, or char
* long to byte, short, char, or int
* float to byte, short, char, int, or long
* double to byte, short, char, int, long, or float

double doubleVal = 9.78;

float floatVal = (float) doubleVal; // double to float

long longVal = (long) floatVal; // float to long

int intVal = (int) longVal; // long to int

short shortVal = (short) intVal; // int to short

byte byteVal = (byte) shortVal; // short to byte

System.out.println("double value: " + doubleVal);

System.out.println("float value: " + floatVal);

System.out.println("long value: " + longVal);

System.out.println("int value: " + intVal);

System.out.println("short value: " + shortVal);

System.out.println("byte value: " + byteVal);

**3. Casting Between Objects (Reference Type Casting)**

In Java, reference type casting applies to objects and can be either **upcasting** or **downcasting**.

* **Upcasting:** Casting a subclass object to a superclass type. This is done automatically and does not require explicit casting.
* **Downcasting:** Casting a superclass object back to a subclass type. This requires explicit casting and can throw a ClassCastException if not done correctly.

**Example of Upcasting and Downcasting:**

class Animal {

void makeSound() {

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

}

}

class Dog extends Animal {

void makeSound() {

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

}

void fetch() {

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

}

}

public class Main {

public static void main(String[] args) {

Animal myAnimal = new Dog(); // Upcasting

myAnimal.makeSound(); // Calls the Dog's method

Dog myDog = (Dog) myAnimal; // Downcasting

myDog.fetch(); // Calls the fetch method of Dog class

}

}

1. **What is Automatic Type Promotion in Java?**

Automatic type promotion in Java refers to the process by which smaller data types are automatically converted (promoted) to larger data types when performing operations involving different data types. This promotion ensures that the operation is performed safely without losing data.

**Key Points about Automatic Type Promotion:**

1. **Expression Evaluation:**
   * When performing arithmetic operations or expressions involving multiple data types, Java promotes smaller types to larger types to prevent data loss.
   * For example, if an operation involves an int and a double, the int is promoted to a double before the operation is performed.
2. **Promotion Rules:**
   * **Byte, Short, and Char to Int:** When a byte, short, or char is used in an arithmetic operation, it is automatically promoted to int before the operation.
   * **Int to Long:** If an int is involved in an operation with a long, the int is promoted to long.
   * **Long to Float:** If an operation involves both long and float, the long is promoted to float.
   * **Float to Double:** If an operation involves both float and double, the float is promoted to double.
3. **Type Promotion in Expressions:**
   * The result of an arithmetic operation follows the type of the largest operand.
   * For example, if you add a byte to an int, the result is an int.

**Example of Automatic Type Promotion:**

**Example 1**

class Main {

public static void main(String[] args) {

byte b = 42;

char c = 'A'; // ASCII value of 'A' is 65

short s = 1024;

int i = 50000;

float f = 5.67f;

double d = .1234;

// In this expression, byte, char, and short are promoted to int

// then the entire expression is promoted to double

double result = (f \* b) + (i / c) - (d \* s);

System.out.println("Result: " + result);

}

}

**Explanation:**

* b (byte) is promoted to int.
* c (char) is promoted to int.
* s (short) is promoted to int.
* The result of (f \* b), (i / c), and (d \* s) is promoted to double because f is float and d is double and in this expression the biggest primitive datatype is double that’s why whole expression is converted to double.

**Output:**

Result: 626.561407079646

**Example 2**

import java.util.Scanner;

public class GradeCalculator {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Number of subjects

int numberOfSubjects = 5;

// Arrays to store subject names and scores

String[] subjects = new String[numberOfSubjects];

int[] scores = new int[numberOfSubjects];

// Input subject names

System.out.println("Enter the names of the subjects:");

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

System.out.print("Subject " + (i + 1) + ": ");

subjects[i] = scanner.nextLine();

}

// Input scores

System.out.println("Enter the scores for each subject (0-100):");

int totalScore = 0;

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

while (true) {

System.out.print(subjects[i] + ": ");

int score = scanner.nextInt();

if (score < 0 || score > 100) {

System.out.println("Invalid score. Please enter a value between 0 and 100.");

} else {

scores[i] = score;

totalScore += score;

break;

}

}

}

// Calculate average score

float averageScore = totalScore / (float)numberOfSubjects;

// Display results

System.out.println("\n--- Results ---");

System.out.println("Total Score: " + totalScore);

System.out.println("Average Score: " + averageScore);

// Provide feedback based on average score

if (averageScore >= 90) {

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

System.out.println("Excellent work!");

} else if (averageScore >= 80) {

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

System.out.println("Good job!");

} else if (averageScore >= 70) {

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

System.out.println("Fair performance.");

} else if (averageScore >= 60) {

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

System.out.println("Needs improvement.");

} else {

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

System.out.println("Fail. Please seek help and try again.");

}

scanner.close();

}

}

**Explanation**

1. **Imports and Setup:**
   * The program imports java.util.Scanner for user input and initializes the Scanner object.
2. **Subject and Score Arrays:**
   * Arrays subjects and scores are used to store the names of subjects and the corresponding scores.
3. **User Input for Subjects:**
   * The program prompts the user to input names for a predefined number of subjects.
4. **User Input for Scores:**
   * The program collects scores for each subject, ensuring the scores are within the valid range (0-100). It uses a while loop to validate input.
5. **Average Calculation:**
   * The average score is calculated by dividing the total score by the number of subjects. Note the use of (float)numberOfSubjects to ensure the division is performed with floating-point precision.
6. **Feedback Based on Average:**
   * The program provides feedback based on the average score, assigning grades and comments.
7. **Output:**
   * The program prints the total and average scores, and provides feedback based on the performance.

**Key Concepts Illustrated**

* Automatic Type Promotion: Demonstrated in the calculation of the average score.
* User Input and Validation: Collecting and validating user input using loops and conditionals.
* Arrays: Storing and accessing multiple values efficiently.
* Conditionals: Providing feedback based on different ranges of average scores.

# Operators in java

**What is an Operator in Java?**

In Java, an operator is a symbol that performs operations on one or more operands (variables, values, or expressions). Operators are essential in programming as they allow you to manipulate data and variables, perform calculations, and control the flow of logic within a program.

* **Operator**: it is a symbol which perform specific task. Ex +,-,\*,/
* **Operands**: it is a simple variable which holds mathematical value. {Age = 23;}

**Why Are Operators Important?**

* **Data Manipulation:** Operators allow you to perform mathematical operations, compare values, and manipulate variables, enabling the processing of data.
* **Control Flow:** Logical and relational operators help in decision-making and control the flow of the program by evaluating conditions.
* **Efficient Coding:** Operators simplify code by providing concise ways to perform operations that would otherwise require longer, more complex expressions.

**Classification of Operators in Java**

Java operators can be classified into three main categories:

1. **Unary Operators**
2. **Binary Operators**
3. **Ternary Operators**

**1. Unary Operators**

Unary operators operate on a single operand. They typically perform operations like incrementing/decrementing a value, negating a boolean value, or inverting the sign of a number.

**Example of Unary Operators:**

int a = 10;

int b = -a; // Unary minus, b becomes -10

a++; // Increment, a becomes 11

boolean flag = true;

flag = !flag; // Logical NOT, flag becomes false

**2. Binary Operators**

Binary operators operate on two operands. These are the most commonly used operators in Java and are divided into several categories.

**Example of Binary Operators:**

int a = 10;

int b = 5;

int sum = a + b; // Addition

boolean isEqual = (a == b); // Relational operator, isEqual becomes false

int andResult = a & b; // Bitwise AND

a += b; // Add and assign, a becomes 15

**3. Ternary Operator**

The ternary operator is the only operator in Java that takes three operands. It is a shorthand for if-else statements and is used to evaluate a boolean expression and return one of two values based on the result.

**Ternary Operator:**

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

**Example of Ternary Operator:**

int a = 10;

int b = 20;

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

**Here is the table which give you clear picture of Operators.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Category | Operator | | Definition | Example |
| Unary |  | |  |  |
|  | + | | Unary plus operator. Indicates a positive value. | +5 |
|  | - | | Unary minus operator. Negates the value. | -5 |
|  | ++ | | Increment operator. Increases the value of the variable by 1. | i++ (post-increment) |
|  | -- | | Decrement operator. Decreases the value of the variable by 1. | i-- (post-decrement) |
|  | ! | | Logical NOT operator. Inverts the boolean value. | !true (results in false) |
|  | ~ | | Bitwise NOT operator. Inverts each bit of the operand. | ~0 (results in -1) |
|  |  | |  |  |
| Binary |  | |  |  |
|  | **Arithmetic** | |  |  |
|  | + | | Addition operator. Adds two values. | 5 + 3 (results in 8) |
|  | - | | Subtraction operator. Subtracts the second value from the first. | 5 - 3 (results in 2) |
|  | \* | | Multiplication operator. Multiplies two values. | 5 \* 3 (results in 15) |
|  | / | | Division operator. Divides the first value by the second. | 6 / 3 (results in 2) |
|  | % | | Modulus operator. Returns the remainder of the division. | 5 % 3 (results in 2) |
|  |  | |  |  |
| Binary |  | |  |  |
|  | **Relational** | |  |  |
|  | == | | Equality operator. Checks if two values are equal. | 5 == 5 (results in true) |
|  | != | | Inequality operator. Checks if two values are not equal. | 5 != 3 (results in true) |
|  | > | | Greater than operator. Checks if the first value is greater than the second. | 5 > 3 (results in true) |
|  | < | | Less than operator. Checks if the first value is less than the second. | 5 < 3 (results in false) |
|  | >= | | Greater than or equal to operator. Checks if the first value is greater than or equal to the second. | 5 >= 5 (results in true) |
|  | <= | | Less than or equal to operator. Checks if the first value is less than or equal to the second. | 5 <= 3 (results in false) |
|  |  | |  |  |
| Binary |  | |  |  |
|  | **Logical** | |  |  |
|  | && | | Logical AND operator. Returns true if both operands are true. | true && false (results in false) |
|  | ` | |  | ` |
|  | ^ | | Logical XOR operator. Returns true if exactly one operand is true. | true ^ false (results in true) |
|  |  | |  |  |
| Binary |  | |  |  |
|  | **Bitwise** | |  |  |
|  | & | | Bitwise AND operator. Performs a bitwise AND operation. | 5 & 3 (results in 1) |
|  | ^ | | Bitwise XOR operator. Performs a bitwise XOR operation. | 5 ^ 3 (results in 6) |
|  | << | | Left shift operator. Shifts bits to the left, filling with zeroes. | 5 << 1 (results in 10) |
|  | >> | | Right shift operator. Shifts bits to the right, preserving the sign. | 5 >> 1 (results in 2) |
|  | >>> | | Unsigned right shift operator. Shifts bits to the right, filling with zeroes regardless of the sign. | 5 >>> 1 (results in 2) |
|  |  | |  |  |
| Binary |  | |  |  |
|  | **Assignment** | |  |  |
|  | = | | Assignment operator. Assigns a value to a variable. | x = 10 |
|  | += | | Addition assignment operator. Adds and assigns the result to the variable. | x += 5 (equivalent to x = x + 5) |
|  | -= | | Subtraction assignment operator. Subtracts and assigns the result to the variable. | x -= 5 (equivalent to x = x - 5) |
|  | \*= | | Multiplication assignment operator. Multiplies and assigns the result to the variable. | x \*= 5 (equivalent to x = x \* 5) |
|  | /= | | Division assignment operator. Divides and assigns the result to the variable. | x /= 5 (equivalent to x = x / 5) |
|  | %= | | Modulus assignment operator. Takes modulus and assigns the result to the variable. | x %= 5 (equivalent to x = x % 5) |
|  |  | |  |  |
| Ternary |  | |  |  |
|  | ?: | | Conditional (ternary) operator. Returns one of two values based on a boolean expression. | a > b ? a : b (returns a if a > b, otherwise b) |
|  |  | |  |  |
| Java Special Operator | | |  |  |
|  | | |  |  |
| Category | | **Operator** |  |  |
| Type checking | | instanceof | Tests whether an object is an instance of a specific class or subclass. | if (obj instanceof String) (true if obj is an instance of String) |
| Type casting | | (Type) | Explicitly converts a variable from one type to another. | String s = (String) obj; (casts obj to String) |
| Method reference | | :: | Method reference operator. Provides a way to refer to methods without invoking them. | List<String> list = Arrays.asList("a", "b", "c"); list.forEach(System.out::println); |
| Object Creator | | new | Creates new objects or arrays. | String s = new String("Hello"); |
| Member Access | | . | Member access operator. Used to access fields and methods of objects. | obj.method(); (calls method on obj) |
| Annotation | | @ | Annotation. Used to provide metadata about the program elements (e.g., classes, methods). | @Override public void method() {} |
|  |  | |  |  |

**All the details of each operator is mandatory after the notes completion**

**Notations for unary operator**

|  |  |  |  |
| --- | --- | --- | --- |
| Notations |  |  |  |
| prefix | The prefix increment (++i) and decrement (--i) operators increase or decrease the value of a variable by 1, respectively. The key characteristic of prefix operators is that they perform the operation **before** the current value is used in an expression. | | ++i, --i  int i = 5;  int result = ++i; |
| Infix | Infix operators are placed between two operands and are used to perform operations on those operands. They are evaluated based on their precedence and associativity rules. | | int sum = 5 + 3;  int difference = 5 - 3;  int quotient = 6 / 3;  int remainder = 5 % 3; |
| postfix | The postfix increment (i++) and decrement (i--) operators increase or decrease the value of a variable by 1, respectively. The key characteristic of postfix operators is that they perform the operation **after** the current value has been used in an expression. | | i--, i++  int i = 5;  int result = i++; |

# Literals

In Java, literals are the fixed values used directly in the code to represent data. They are the simplest form of data, used to initialize variables, define constants, and perform operations. Literals in Java can be categorized based on the type of data they represent. Here’s a detailed explanation of all types of literals available in Java:

**1. Integer Literals**

**Definition**: Represent whole numbers. They can be specified in decimal, octal, hexadecimal, or binary formats.

* **Decimal Literals**: Base-10 numbers. They are the most common form of integer literals.

int decimal = 42; // Decimal literal

* **Octal Literals**: Base-8 numbers, prefixed with 0.

int octal = 052; // Octal literal (equal to decimal 42)

* **Hexadecimal Literals**: Base-16 numbers, prefixed with 0x or 0X.

int hex = 0x2A; // Hexadecimal literal (equal to decimal 42)

* **Binary Literals**: Base-2 numbers, prefixed with 0b or 0B. Available from Java 7 onwards.

int binary = 0b101010; // Binary literal (equal to decimal 42)

**Examples**:

int decimal = 100; // Decimal representation

int octal = 0144; // Octal representation (equal to decimal 100)

int hex = 0x64; // Hexadecimal representation (equal to decimal 100)

int binary = 0b1100100; // Binary representation (equal to decimal 100)

**2. Floating-Point Literals**

**Definition**: Represent numbers with a fractional part. They are used to define floating-point numbers.

* **Float Literals**: Represent single-precision floating-point numbers. They are suffixed with f or F.

float floatValue = 3.14f; // Float literal

* **Double Literals**: Represent double-precision floating-point numbers. They are the default type for floating-point literals and can be suffixed with d or D, though the suffix is optional.

double doubleValue = 3.14159; // Double literal

**Examples**:

float f = 1.23f; // Float literal with 'f' suffix

double d = 1.234567; // Double literal (no suffix needed)

**3. Character Literals**

**Definition**: Represent single characters. They are enclosed in single quotes.

* **Single Characters**: Can include letters, digits, special characters, and escape sequences.

char letter = 'A'; // Character literal

char digit = '7'; // Character literal

char special = '#'; // Character literal

char escape = '\n'; // Escape sequence for new line

**4. String Literals**

**Definition**: Represent sequences of characters enclosed in double quotes. They can include any printable characters and escape sequences.

* **Simple Strings**: A sequence of characters.

String hello = "Hello, World!"; // String literal

* **Escape Sequences**: Special characters within strings, like newline (\n), tab (\t), and double quote (\").

String escape = "Line 1\nLine 2"; // Newline character within the string

String quote = "He said, \"Hello!\""; // Double quotes within the string

**Examples**:

String str1 = "Hello"; // Simple string

String str2 = "Hello\nWorld"; // String with newline

String str3 = "He said, \"Hi!\""; // String with escaped double quotes

**5. Boolean Literals**

**Definition**: Represent true or false values. They are used in conditional statements and logical operations.

* **True Literal**: Represents the boolean value true.

boolean isTrue = true; // Boolean literal

* **False Literal**: Represents the boolean value false.

boolean isFalse = false; // Boolean literal

**Examples**:

boolean flag = true; // Boolean true

boolean isValid = false; // Boolean false

**Literal Representation in table**

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Format | Example | Description |
| Integer | Decimal | 42 | Base-10 integer |
|  | Octal | 052 | Base-8 integer (prefix 0) |
|  | Hexadecimal | 0x2A | Base-16 integer (prefix 0x) |
|  | Binary | 0b101010 | Base-2 integer (prefix 0b) |
|  |  |  |  |
| Floating-Point | Float | 3.14f | Single-precision floating-point (suffix f) |
|  | Double | 3.14159 | Double-precision floating-point |
|  |  |  |  |
| Character | Single Character | 'A' | Single character enclosed in single quotes |
|  | Escape Sequence | '\n' | Special characters within single quotes |
|  | Unicode | '\u0041' | Unicode character representation |
|  |  |  |  |
| String | Sequence of Characters | "Hello" | String enclosed in double quotes |
|  | Escape Sequences | "Line 1\nLine 2" | Special characters within the string |
|  |  |  |  |
| Boolean | True | true | Boolean true |
|  | False | false | Boolean false |

**Escape Sequence in java**

In Java, escape sequences are used to represent special characters within string literals that would otherwise be difficult to include directly. These sequences start with a backslash (\) followed by a specific character, which the Java compiler interprets in a special way.

Here are the escape sequences in Java along with their definitions:

**Java Escape Sequences**

|  |  |
| --- | --- |
| Escape Sequence | Definition |
| \b | **Backspace**: Moves the cursor back one position (erases the previous character). |
| \f | **Form Feed**: Moves the cursor to the beginning of the next page (mostly used in printing contexts). |
| \n | **New Line**: Moves the cursor to the beginning of the next line (Line Feed). |
| \r | **Carriage Return**: Moves the cursor to the beginning of the current line (does not advance to the next line). |
| \t | **Tab**: Inserts a horizontal tab (a space of a tab width). |
| \" | **Double Quote**: Inserts a double quote character inside a string literal. |
| \' | **Single Quote**: Inserts a single quote character inside a character literal. |
| \\ | **Backslash**: Inserts a backslash character (\) into a string literal. |
| \uXXXX | **Unicode Character**: Inserts a Unicode character where XXXX is the four-digit hexadecimal code of the character. |

**Working Definitions**

1. **Backspace (\b)**:
   * **Usage**: Removes the character before the cursor position in a string. Often used in console applications to modify output dynamically.
2. **Form Feed (\f)**:
   * **Usage**: Advances the output to the next page in printing contexts. Its practical use is rare in modern applications, but it is part of the ASCII standard.
3. **New Line (\n)**:
   * **Usage**: Moves the cursor to the next line. It is commonly used to format output across multiple lines.
4. **Carriage Return (\r)**:
   * **Usage**: Moves the cursor to the start of the current line. Often used in conjunction with \n to start a new line on some platforms (e.g., \r\n on Windows).
5. **Tab (\t)**:
   * **Usage**: Inserts a horizontal tab, creating a space equivalent to a tab width. Useful for aligning text in output.
6. **Double Quote (\")**:
   * **Usage**: Allows the inclusion of double quotes within a string literal. Necessary for string literals that include quotes.
7. **Single Quote (\')**:
   * **Usage**: Allows the inclusion of a single quote in a character literal. This is particularly used in single-quoted characters.
8. **Backslash (\\)**:
   * **Usage**: Allows the inclusion of a literal backslash in a string. Essential when a backslash needs to appear in the output.
9. **Unicode Character (\uXXXX)**:
   * **Usage**: Inserts a Unicode character specified by the four-digit hexadecimal code. Useful for representing characters from various languages and symbols not available on a standard keyboard.

**Example Usage**

Here’s how some of these escape sequences might appear in a Java program:

public class EscapeSequenceExample {

public static void main(String[] args) {

System.out.println("This is a tab:\tEnd of tab.");

System.out.println("This is a new line:\nNext line.");

System.out.println("This is a backslash: \\");

System.out.println("This is a double quote: \"");

System.out.println("This is a Unicode character: \u0041"); // Prints 'A'

}

}

In this example:

* \t inserts a tab space.
* \n inserts a newline.
* \\ inserts a backslash.
* \" inserts a double quote.
* \u0041 represents the Unicode character for 'A'.

# Keywords

In Java, **keywords** are reserved words that have predefined meanings in the language syntax. They are fundamental to Java programming and cannot be used as identifiers (such as variable names, function names, or class names). Each keyword has a specific purpose and role in the language. Here’s a detailed explanation of each Java keyword and its usage:

|  |  |  |
| --- | --- | --- |
| Category | Keyword | Description |
|  |  |  |
| Access Control | private | Defines a member of a class that is accessible only within the class. |
|  | protected | Defines a member that is accessible within the same package and by subclasses. |
|  | public | Defines a member that is accessible from any other class. |
|  |  |  |
| Class and Object | abstract | Declares a class or method that cannot be instantiated or must be implemented by subclasses. |
|  | class | Defines a new class. |
|  | interface | Defines an interface. |
|  | extends | Indicates that a class or interface inherits from another class or interface. |
|  | implements | Indicates that a class implements an interface. |
|  | final | Defines constants, prevents method overriding, and prevents class inheritance. |
|  | static | Defines a member that belongs to the class rather than instances. |
|  | this | Refers to the current instance of a class. |
|  | super | Refers to the superclass of the current object, used to access superclass methods and constructors. |
|  | record | Defines a class that acts as a transparent carrier for immutable data. |
|  | sealed | Restricts which classes can extend a class or implement an interface. |
|  | non-sealed | Allows a sealed class to be extended. |
|  | transient | Prevents serialization of a member variable. |
|  |  |  |
| Control Flow | break | Exits from the current loop or switch statement prematurely. |
|  | continue | Skips the current iteration of a loop and proceeds to the next iteration. |
|  | default | Defines a default block of code in a switch statement to execute if no case matches. |
|  | do | Defines a loop that executes a block of code once before checking the loop’s condition. |
|  | else | Defines a block of code to execute if the if condition evaluates to false. |
|  | for | Defines a loop that executes a block of code a specific number of times. |
|  | if | Defines a conditional statement that executes a block of code if its condition evaluates to true. |
|  | switch | Defines a multi-way branch statement based on the value of an expression. |
|  | while | Defines a loop that executes as long as its condition evaluates to true. |
|  | yield | Exits from a switch expression and returns a value. |
|  |  |  |
| Exception Handling | try | Defines a block of code that is tested for exceptions. |
|  | catch | Defines a block of code that handles exceptions thrown by a try block. |
|  | finally | Defines a block of code that always executes after a try-catch block, regardless of exceptions. |
|  | throw | Explicitly throws an exception. |
|  | throws | Specifies the exceptions that a method might throw. |
|  |  |  |
| Data Types | byte | Defines a variable of type byte (8-bit signed integer). |
|  | short | Defines a variable of type short (16-bit signed integer). |
|  | int | Defines a variable of type int (32-bit signed integer). |
|  | long | Defines a variable of type long (64-bit signed integer). |
|  | float | Defines a variable of type float (32-bit floating-point number). |
|  | double | Defines a variable of type double (64-bit floating-point number). |
|  | char | Defines a variable of type char (16-bit Unicode character). |
|  | boolean | Defines a variable of type boolean (true or false). |
|  |  |  |
| Package Management | import | Imports classes or entire packages into the current file. |
|  | package | Defines a namespace for organizing classes and interfaces. |
|  |  |  |
| Modules and Packages | module | Defines a module that groups related packages and resources. |
|  | requires | Specifies a dependency on another module. |
|  | exports | Specifies which packages are accessible from other modules. |
|  | opens | Allows other modules to access the internal packages at runtime. |
|  | uses | Specifies a service provider that is used by a module. |
|  | provides | Specifies a service implementation provided by a module. |
|  | permits | Defines permitted subclasses in a sealed class. |
|  | to | Used in records to specify component names. |
|  | with | Used in records to specify components in pattern matching. |
|  |  |  |
| Concurrency | synchronized | Ensures that a block of code or method is accessed by only one thread at a time. |
|  | volatile | Indicates that a variable’s value may be modified by different threads. |
|  |  |  |
| Other | assert | Used for debugging purposes to test assumptions about the program’s state during runtime. |
|  | const | Reserved for future use; not currently used in Java. |
|  | goto | Reserved for future use; not currently used in Java. |
|  | native | Specifies that a method is implemented in native code using JNI (Java Native Interface). |
|  | null | Represents a null reference, indicating that a variable does not refer to any object. |
|  | return | Exits from the current method and optionally returns a value. |
|  | void | Specifies that a method does not return any value. |
|  | strictfp | Ensures floating-point calculations adhere strictly to IEEE 754 standards. |
|  | transitive | Allows transitive dependencies in module definitions. |
|  | var | Local variable type inference for declaring variables. |
|  | when | Used in pattern matching with records (Java 17+). |

# Identifiers

In Java, **identifiers** are names given to various program elements such as classes, methods, variables, and other entities. Identifiers serve as the means to uniquely identify these elements in the code. They play a crucial role in defining and manipulating objects and functions within a Java program.

**Types of Identifiers in Java**

Here is a detailed explanation of the types of identifiers in Java:

1. **Class Identifiers**
   * **Definition**: Names given to classes. A class identifier is used to create and refer to objects of that class.
   * **Example**: MyClass, AccountManager, StudentRecord.
2. **Method Identifiers**
   * **Definition**: Names given to methods within a class. Methods define the behavior of objects created from the class.
   * **Example**: calculateTotal(), printDetails(), updateRecord().
3. **Variable Identifiers**
   * **Definition**: Names given to variables, which hold data values. Variables can be local, instance, or class (static) variables.
   * **Example**: totalAmount, studentName, isActive.
4. **Constant Identifiers**
   * **Definition**: Names given to constants, which are variables whose values cannot be changed after initialization. Constants are usually declared using the final keyword.
   * **Example**: MAX\_VALUE, PI, DEFAULT\_TIMEOUT.
5. **Package Identifiers**
   * **Definition**: Names given to packages, which are used to group related classes and interfaces.
   * **Example**: java.util, com.example.project, org.apache.commons.
6. **Interface Identifiers**
   * **Definition**: Names given to interfaces, which define a contract that classes can implement.
   * **Example**: Serializable, Cloneable, Runnable.
7. **Enum Identifiers**
   * **Definition**: Names given to enumerations, which define a set of named constants.
   * **Example**: DayOfWeek, Color, Direction.

**Rules for Identifiers**

Java identifiers must follow specific rules:

1. **Start with a Letter, Underscore, or Dollar Sign**: Identifiers must begin with a letter (a-z, A-Z), an underscore (\_), or a dollar sign ($). They cannot start with a digit.
   * Valid: studentName, \_age, $salary
   * Invalid: 1stName, #number
2. **Subsequent Characters**: After the first character, identifiers can include letters, digits (0-9), underscores (\_), and dollar signs ($).
   * Valid: totalAmount2, employee\_name, data$price
3. **Case Sensitivity**: Identifiers are case-sensitive. This means that MyClass, myclass, and MYCLASS are considered different identifiers.
   * Example: myVariable, MyVariable
4. **No Reserved Keywords**: Identifiers cannot be Java reserved keywords.
   * Invalid: class, int, public
5. **Cannot be Java Primitive Types**: Identifiers cannot be Java primitive types like int, float, char.
   * Invalid: int, float
6. **No Special Characters**: Identifiers cannot contain special characters like @, #, !, etc.
   * Invalid: student-name, total@amount

**Best Practices for Naming Identifiers**

1. **Meaningful Names**: Use descriptive names that convey the purpose of the identifier.
   * Good: calculateTotalAmount, studentGrade
   * Bad: x, temp
2. **Consistent Naming Conventions**:
   * **Classes**: Use CamelCase (e.g., AccountManager, StudentRecord).
   * **Methods and Variables**: Use camelCase (e.g., calculateTotal(), studentName).
   * **Constants**: Use UPPER\_SNAKE\_CASE (e.g., MAX\_VALUE, DEFAULT\_TIMEOUT).
3. **Avoid Single Letters**: Except for loop counters, single-letter identifiers are generally discouraged as they are not descriptive.
   * Example: Use counter instead of c for better readability.
4. **Avoid Magic Numbers**: Instead of using raw numbers, define them as constants with meaningful names.
   * Good: MAX\_RETRIES
   * Bad: 5

**Valid Identifiers**

* TestVariable
* testvariable
* a
* i
* Test\_Variable
* \_testvariable
* $testvariable
* sum\_of\_array
* TESTVARIABLE
* jtp123
* JavaTpoint
* Javatpoint123

**Invalid Identifiers / Bad identifiers**

* Test Variable ( We can not include a space in an identifier)
* 123javatpoint ( The identifier should not begin with numbers)
* java+tpoint ( The plus (+) symbol cannot be used)
* a-javatpoint ( Hyphen symbol is not allowed)
* java\_&\_Tpoint ( ampersand symbol is not allowed)
* Java'tpoint (we can not use an apostrophe symbol in an identifier)

By following these guidelines, you ensure that your code is more readable, maintainable, and less error-prone. Identifiers are a fundamental part of Java programming, and adhering to these rules and best practices will help in writing clean and effective code.

# Control statement in java

**Comprehensive Guide to Java Control Statements**

In Java, control statements are essential constructs that determine the flow of execution in a program. They allow developers to control the order in which code is executed, based on certain conditions or repetitive tasks. This detailed guide will explore all Java control statements, providing in-depth explanations and practical examples.

**I. Conditional Statements**

Conditional statements allow you to execute different blocks of code based on certain conditions. Java provides several conditional statements, including if, else, else if, and switch.

**A. if Statement**

The if statement evaluates a boolean expression and executes a block of code if the condition is true.

**Syntax:**

if (condition) {

// Code to execute if condition is true

}

**Example:**

public class IfExample {

public static void main(String[] args) {

int number = 10;

if (number > 0) {

System.out.println("The number is positive.");

}

}

}

In this example, the condition number > 0 is true, so the message "The number is positive." is printed.

**B. else Statement**

The else statement follows an if statement and executes a block of code if the if condition is false.

**Syntax:**

if (condition) {

// Code to execute if condition is true

} else {

// Code to execute if condition is false

}

**Example:**

public class ElseExample {

public static void main(String[] args) {

int number = -5;

if (number > 0) {

System.out.println("The number is positive.");

} else {

System.out.println("The number is not positive.");

}

}

}

Here, since number is not greater than 0, the program prints "The number is not positive."

**C. else if Statement**

The else if statement allows you to test multiple conditions sequentially.

**Syntax:**

if (condition1) {

// Code to execute if condition1 is true

} else if (condition2) {

// Code to execute if condition2 is true

} else {

// Code to execute if none of the conditions are true

}

**Example:**

public class ElseIfExample {

public static void main(String[] args) {

int number = 0;

if (number > 0) {

System.out.println("The number is positive.");

} else if (number < 0) {

System.out.println("The number is negative.");

} else {

System.out.println("The number is zero.");

}

}

}

This example checks if the number is positive, negative, or zero, and prints the appropriate message.

**D. switch Statement**

The switch statement is used to execute one block of code among many options based on the value of an expression. It is more readable than a series of if-else statements when dealing with multiple conditions.

**Syntax:**

switch (expression) {

case value1:

// Code to execute if expression equals value1

break;

case value2:

// Code to execute if expression equals value2

break;

default:

// Code to execute if expression does not match any value

}

**Example:**

public class SwitchExample {

public static void main(String[] args) {

int day = 3;

switch (day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

case 3:

System.out.println("Wednesday");

break;

case 4:

System.out.println("Thursday");

break;

case 5:

System.out.println("Friday");

break;

case 6:

System.out.println("Saturday");

break;

case 7:

System.out.println("Sunday");

break;

default:

System.out.println("Invalid day");

}

}

}

In this example, the switch statement prints "Wednesday" because day is 3.

**II. Looping Statements**

Looping statements allow you to execute a block of code repeatedly as long as a specified condition is true. Java provides several looping constructs: for, while, and do-while.

**A. for Loop**

The for loop is used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and iteration.

**Syntax:**

for (initialization; condition; iteration) {

// Code to execute repeatedly

}

**Example:**

public class ForLoopExample {

public static void main(String[] args) {

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

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

}

}

}

This for loop prints the values of i from 0 to 4.

**Real world example**

public class MultiplicationTable {

public static void main(String[] args) {

int size = 10; // Size of the multiplication table

System.out.println("Multiplication Table up to " + size + ":");

// Generate the multiplication table using a for loop

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

for (int j = 1; j <= size; j++) {

System.out.printf("%4d", i \* j); // Print each value, formatted

}

System.out.println(); // Move to the next line after each row

}

}

}

**Explanation:**

* The outer for loop iterates through the rows of the table.
* The inner for loop iterates through the columns for each row.
* This example uses printf to format the output for better readability.

**B. while Loop**

The while loop executes a block of code as long as the specified condition is true. The condition is evaluated before the code block is executed.

**Syntax:**

while (condition) {

// Code to execute repeatedly

}

**Example:**

public class WhileLoopExample {

public static void main(String[] args) {

int i = 0;

while (i < 5) {

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

i++;

}

}

}

Here, the while loop prints the values of i from 0 to 4.

**Real world example**

import java.util.Random;

import java.util.Scanner;

public class NumberGuessingGame {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

Random random = new Random();

// Generate a random number between 1 and 100

int targetNumber = random.nextInt(100) + 1;

int guess = 0; // User's guess

int numberOfAttempts = 0;

System.out.println("Welcome to the Number Guessing Game!");

System.out.println("I have picked a number between 1 and 100. Try to guess it.");

// Loop until the user guesses the correct number

while (guess != targetNumber) {

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

guess = scanner.nextInt();

numberOfAttempts++;

// Provide feedback based on the user's guess

if (guess < targetNumber) {

System.out.println("Too low! Try again.");

} else if (guess > targetNumber) {

System.out.println("Too high! Try again.");

} else {

System.out.println("Congratulations! You've guessed the number.");

System.out.println("It took you " + numberOfAttempts + " attempts.");

}

}

scanner.close();

}

}

**Explanation**

1. **Random Number Generation:**
   * The Random class is used to generate a random number between 1 and 100. This number is the target that the user needs to guess.
2. **User Input:**
   * The program prompts the user to enter their guess. This is done within the while loop, which continues until the user guesses the correct number.
3. **Feedback and Attempt Counting:**
   * The program provides feedback based on whether the user's guess is too low or too high.
   * The number of attempts is counted and displayed once the correct number is guessed.
4. **Loop Condition:**
   * The while loop continues running as long as the guess is not equal to the targetNumber. This ensures that the user keeps guessing until they find the correct number.

**C. do-while Loop**

The do-while loop is similar to the while loop, but the condition is evaluated after the block of code has been executed. This ensures that the code block is executed at least once.

**Syntax:**

do {

// Code to execute repeatedly

} while (condition);

**Example:**

public class DoWhileLoopExample {

public static void main(String[] args) {

int i = 0;

do {

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

i++;

} while (i < 5);

}

}

In this example, the do-while loop prints the values of i from 0 to 4, similar to the while loop, but ensures the code block runs at least once.

**Real world example 1**

import java.util.Scanner;

public class ATMWithdrawal {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int balance = 1000; // Starting balance

int choice;

do {

// Display menu

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

System.out.println("1. Check Balance");

System.out.println("2. Withdraw Money");

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

System.out.print("Choose an option (1-3): ");

choice = scanner.nextInt();

// Handle user choice

switch (choice) {

case 1:

System.out.println("Current Balance: $" + balance);

break;

case 2:

System.out.print("Enter amount to withdraw: $");

int amount = scanner.nextInt();

if (amount > balance) {

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

} else {

balance -= amount;

System.out.println("Withdrawal successful. New Balance: $" + balance);

}

break;

case 3:

System.out.println("Exiting. Have a nice day!");

break;

default:

System.out.println("Invalid choice. Please choose a valid option.");

}

} while (choice != 3); // Repeat until the user chooses to exit

scanner.close();

}

}

**Explanation:**

* The while loop continues to display the ATM menu and process user input until the user selects the option to exit.
* The do-while loop ensures that the menu is displayed at least once and the program continues to prompt the user until they choose to exit.

**Real world example 2**

import java.util.Scanner;

public class PositiveNumberValidator {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int number;

// Prompt the user to enter a positive number

do {

System.out.print("Enter a positive number: ");

number = scanner.nextInt();

if (number <= 0) {

System.out.println("The number must be positive. Please try again.");

}

} while (number <= 0); // Continue until a positive number is entered

System.out.println("Thank you! You entered a positive number: " + number);

scanner.close();

}

}

**Explanation:**

* The do-while loop prompts the user to enter a positive number.
* The loop guarantees that the prompt will execute at least once and will continue to ask for input until the user provides a valid positive number.

**III. Jump Statements**

Jump statements alter the flow of control in a program. Java provides break, continue, and return as jump statements.

**A. break Statement**

The break statement is used to exit a loop or switch statement prematurely.

**Syntax:**

break;

**Example:**

public class BreakExample {

public static void main(String[] args) {

for (int i = 0; i < 10; i++) {

if (i == 5) {

break;

}

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

}

}

}

In this example, the break statement exits the for loop when i equals 5, so only values from 0 to 4 are printed.

**Real world example**

import java.util.ArrayList;

import java.util.List;

public class ItemSearch {

public static void main(String[] args) {

// List of items

List<String> items = new ArrayList<>();

items.add("apple");

items.add("banana");

items.add("cherry");

items.add("date");

items.add("elderberry");

String target = "cherry";

boolean found = false;

// Search for the target item

for (String item : items) {

if (item.equals(target)) {

System.out.println("Item found: " + item);

found = true;

break; // Exit the loop early

}

}

if (!found) {

System.out.println("Item not found.");

}

}

}

**Explanation:**

* break is used to exit the for loop as soon as the target item ("cherry") is found, preventing further unnecessary iterations.

**B. continue Statement**

The continue statement skips the current iteration of a loop and proceeds with the next iteration.

**Syntax:**

continue;

**Example:**

public class ContinueExample {

public static void main(String[] args) {

for (int i = 0; i < 10; i++) {

if (i % 2 == 0) {

continue;

}

System.out.println("Odd value of i: " + i);

}

}

}

Here, the continue statement skips even values of i, printing only the odd numbers between 0 and 9.

**Real world example**

import java.util.ArrayList;

import java.util.List;

public class EmailProcessor {

public static void main(String[] args) {

// List of email addresses (some valid, some invalid)

List<String> emailList = new ArrayList<>();

emailList.add("john.doe@example.com");

emailList.add("jane.doe@com");

emailList.add("invalid-email");

emailList.add("mary.smith@example.org");

emailList.add("another-invalid-email@");

// Process emails

System.out.println("Processing valid email addresses:");

for (String email : emailList) {

// Check if the email is valid

if (!isValidEmail(email)) {

// Skip invalid email addresses

continue;

}

// Process valid email address (e.g., print it out)

System.out.println(email);

}

}

// Method to check if an email address is valid

public static boolean isValidEmail(String email) {

// Simple validation: must contain '@' and '.'

return email.contains("@") && email.contains(".");

}

}

**Explanation**

1. **List of Emails:**
   * We create a list of email addresses that contains both valid and invalid entries.
2. **Processing Loop:**
   * The for loop iterates over each email address in the list.
3. **Validation Check:**
   * The **isValidEmail** method is used to validate each email address. This method performs a simple check to ensure the email contains both @ and . characters.
4. **Skipping Invalid Emails:**
   * If an email address is invalid (**isValidEmail** returns false), the continue statement is used to skip the rest of the loop body and move to the next email in the list.
5. **Processing Valid Emails:**
   * Only valid email addresses are processed (in this case, simply printed out).

**C. return Statement**

The return statement is used to exit from a method and optionally return a value.

**Syntax:**

return; // For methods with no return value

return value; // For methods that return a value

**Example:**

public class ReturnExample {

public static void main(String[] args) {

int result = add(5, 3);

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

}

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

return a + b;

}

}

In this example, the return statement exits the add method and returns the sum of a and b.

**Real world example**

public class FactorialCalculator {

public static void main(String[] args) {

int number = 5;

int result = factorial(number);

System.out.println("Factorial of " + number + " is " + result);

}

// Method to calculate factorial

public static int factorial(int n) {

if (n < 0) {

System.out.println("Factorial is not defined for negative numbers.");

return -1; // Return -1 to indicate an error

}

int result = 1;

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

result \*= i;

}

return result; // Return the factorial result

}

}

**Explanation:**

* **return** is used twice in the factorial method:
  + To exit early with an error code (-1) if the input is negative.
  + To return the computed factorial result if the input is valid.

**Program to demonstrate break, continue and return in one program**

import java.util.ArrayList;

import java.util.List;

class Order {

int id;

boolean isShipped;

boolean isFraudulent;

Order(int id, boolean isShipped, boolean isFraudulent) {

this.id = id;

this.isShipped = isShipped;

this.isFraudulent = isFraudulent;

}

}

public class OrderProcessor {

public static void main(String[] args) {

List<Order> orders = new ArrayList<>();

orders.add(new Order(1, false, false));

orders.add(new Order(2, true, false));

orders.add(new Order(3, false, true));

orders.add(new Order(4, false, false));

orders.add(new Order(5, false, false));

int processedOrders = processOrders(orders);

System.out.println("Total processed orders: " + processedOrders);

}

public static int processOrders(List<Order> orders) {

int processedCount = 0;

for (Order order : orders) {

if (order.isShipped) {

// Skip already shipped orders

continue;

}

if (order.isFraudulent) {

// Stop processing if a fraudulent order is found

System.out.println("Fraudulent order found! Stopping processing.");

break;

}

// Process the order

System.out.println("Processing order ID: " + order.id);

processedCount++;

}

// Return the total number of processed orders

return processedCount;

}

}

1. **Order Class**: This class represents an order with an id, a flag indicating if the order is shipped (isShipped), and a flag indicating if the order is fraudulent (isFraudulent).
2. **OrderProcessor Class**: This class contains the main method and the processOrders method.
3. **Main Method**: Initializes a list of orders and calls the processOrders method, then prints the total number of processed orders.
4. **processOrders Method**:
   * Iterates through the list of orders.
   * Uses continue to skip orders that have already been shipped.
   * Uses break to stop processing if a fraudulent order is found.
   * Processes valid orders by printing their ID and incrementing the processedCount.
   * Returns the total number of processed orders.

This program demonstrates the use of break, continue, and return in a practical scenario involving order processing.

**IV. Nested Control Statements**

Control statements can be nested within each other to create more complex logic. Nested if statements, loops, and switch statements allow for intricate control over program execution.

**A. Nested if Statements**

public class NestedIfExample {

public static void main(String[] args) {

int number = 10;

if (number > 0) {

if (number % 2 == 0) {

System.out.println("The number is positive and even.");

} else {

System.out.println("The number is positive but odd.");

}

} else {

System.out.println("The number is not positive.");

}

}

}

In this example, the outer if statement checks if the number is positive. If true, the nested if statement determines if the number is even or odd. This demonstrates how nested if statements can refine conditions and provide more specific outputs.

**B. Nested Loops**

Nested loops involve placing one loop inside another. This is useful for iterating over multi-dimensional data structures.

**Example:**

public class NestedLoopExample {

public static void main(String[] args) {

for (int i = 1; i <= 3; i++) {

for (int j = 1; j <= 3; j++) {

System.out.println("i = " + i + ", j = " + j);

}

}

}

}

In this example, the outer for loop iterates over the values of i, while the inner for loop iterates over the values of j. This results in a matrix of printed values showing all combinations of i and j.

**C. Nested switch Statements**

switch statements can also be nested to handle more complex branching.

**Example:**

public class NestedSwitchExample {

public static void main(String[] args) {

int day = 2;

int month = 4;

switch (month) {

case 4:

case 6:

case 9:

case 11:

switch (day) {

case 30:

System.out.println("The day is the last day of the month.");

break;

default:

System.out.println("The day is not the last day of the month.");

}

break;

default:

System.out.println("The month is not one with 30 days.");

}

}

}

In this example, the outer switch statement checks the month, and the inner switch statement checks if the day is the last day of that month. This allows for handling multiple levels of conditions.

**Basic Switch case programs**

**Program 1: Nested Switch for Days and Months**

public class NestedSwitchExample1 {

public static void main(String[] args) {

int day = 2;

int month = 4;

switch (month) {

case 4:

case 6:

case 9:

case 11:

switch (day) {

case 30:

System.out.println("The day is the last day of the month.");

break;

default:

System.out.println("The day is not the last day of the month.");

}

break;

default:

System.out.println("The month is not one with 30 days.");

}

}

}

**Program 2: Nested Switch for Vehicle Types and Models**

public class NestedSwitchExample2 {

public static void main(String[] args) {

String vehicleType = "Car";

String vehicleModel = "Sedan";

switch (vehicleType) {

case "Car":

switch (vehicleModel) {

case "Sedan":

System.out.println("The vehicle is a Sedan car.");

break;

case "SUV":

System.out.println("The vehicle is an SUV car.");

break;

default:

System.out.println("Unknown car model.");

}

break;

case "Motorcycle":

System.out.println("The vehicle is a motorcycle.");

break;

default:

System.out.println("Unknown vehicle type.");

}

}

}

**Program 3: Nested Switch for Department and Courses**

public class NestedSwitchExample3 {

public static void main(String[] args) {

String department = "Engineering";

String course = "Computer Science";

switch (department) {

case "Engineering":

switch (course) {

case "Computer Science":

System.out.println("The course is Computer Science in the Engineering department.");

break;

case "Electrical":

System.out.println("The course is Electrical Engineering in the Engineering department.");

break;

default:

System.out.println("Unknown course in the Engineering department.");

}

break;

case "Arts":

System.out.println("The department is Arts.");

break;

default:

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

}

}

}

**4. Nested Switch for Country and State**

public class CountryState {

public static void main(String[] args) {

String country = "USA";

String state = "California";

switch (country) {

case "USA":

System.out.println("Country: USA");

switch (state) {

case "California":

System.out.println("State: California");

break;

case "Texas":

System.out.println("State: Texas");

break;

case "New York":

System.out.println("State: New York");

break;

default:

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

}

break;

case "India":

System.out.println("Country: India");

switch (state) {

case "Maharashtra":

System.out.println("State: Maharashtra");

break;

case "Karnataka":

System.out.println("State: Karnataka");

break;

case "Tamil Nadu":

System.out.println("State: Tamil Nadu");

break;

default:

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

}

break;

default:

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

}

}

}

**Program 5: Switch Inside Switch Inside Switch for Country, State, and City**

public class NestedSwitchExample4 {

public static void main(String[] args) {

String country = "USA";

String state = "California";

String city = "San Francisco";

switch (country) {

case "USA":

switch (state) {

case "California":

switch (city) {

case "San Francisco":

System.out.println("The city is San Francisco in California, USA.");

break;

case "Los Angeles":

System.out.println("The city is Los Angeles in California, USA.");

break;

default:

System.out.println("Unknown city in California, USA.");

}

break;

case "New York":

System.out.println("The state is New York in the USA.");

break;

default:

System.out.println("Unknown state in the USA.");

}

break;

case "Canada":

System.out.println("The country is Canada.");

break;

default:

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

}

}

}

These programs demonstrate different levels of nested switch statements, including one with three levels of nesting.

**Program 6. Nested Switch for Employee Department and Role**

public class EmployeeDepartmentRole {

public static void main(String[] args) {

String department = "IT";

String role = "Developer";

switch (department) {

case "HR":

System.out.println("Department: Human Resources");

switch (role) {

case "Manager":

System.out.println("Role: Manager");

break;

case "Executive":

System.out.println("Role: Executive");

break;

default:

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

}

break;

case "IT":

System.out.println("Department: Information Technology");

switch (role) {

case "Developer":

System.out.println("Role: Developer");

break;

case "Tester":

System.out.println("Role: Tester");

break;

default:

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

}

break;

case "Finance":

System.out.println("Department: Finance");

switch (role) {

case "Analyst":

System.out.println("Role: Analyst");

break;

case "Accountant":

System.out.println("Role: Accountant");

break;

default:

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

}

break;

default:

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

}

}

}

**Program 7. Nested Switch for Vehicle Type and Fuel Type**

public class VehicleTypeFuelType {

public static void main(String[] args) {

String vehicleType = "Car";

String fuelType = "Diesel";

switch (vehicleType) {

case "Car":

System.out.println("Vehicle Type: Car");

switch (fuelType) {

case "Petrol":

System.out.println("Fuel Type: Petrol");

break;

case "Diesel":

System.out.println("Fuel Type: Diesel");

break;

case "Electric":

System.out.println("Fuel Type: Electric");

break;

default:

System.out.println("Unknown Fuel Type");

}

break;

case "Motorcycle":

System.out.println("Vehicle Type: Motorcycle");

switch (fuelType) {

case "Petrol":

System.out.println("Fuel Type: Petrol");

break;

case "Electric":

System.out.println("Fuel Type: Electric");

break;

default:

System.out.println("Unknown Fuel Type");

}

break;

default:

System.out.println("Unknown Vehicle Type");

}

}

}

**Program 8. Nested Switch for Academic Grading System**

public class AcademicGradingSystem {

public static void main(String[] args) {

String grade = "A";

int year = 3;

switch (year) {

case 1:

System.out.println("Freshman Year");

break;

case 2:

System.out.println("Sophomore Year");

break;

case 3:

System.out.println("Junior Year");

switch (grade) {

case "A":

System.out.println("Excellent");

break;

case "B":

System.out.println("Good");

break;

case "C":

System.out.println("Average");

break;

case "D":

System.out.println("Below Average");

break;

default:

System.out.println("Fail");

}

break;

case 4:

System.out.println("Senior Year");

break;

default:

System.out.println("Invalid Year");

}

}

}

**Enhanced Switch Program**

Enhanced switch statements, also known as switch expressions, were introduced in Java 12 as a preview feature and became a standard feature in Java 14. They provide a more concise and flexible syntax for switch statements. Here are some examples of enhanced switch programs

**Example 1: Basic Enhanced Switch**

public class EnhancedSwitchExample1 {

public static void main(String[] args) {

int day = 5;

String dayType = switch (day) {

case 1, 2, 3, 4, 5 -> "Weekday";

case 6, 7 -> "Weekend";

default -> "Invalid day";

};

System.out.println("Day type: " + dayType);

}

}

**Example 2: Enhanced Switch with Multiple Cases**

public class EnhancedSwitchExample2 {

public static void main(String[] args) {

String fruit = "Apple";

int calories = switch (fruit) {

case "Apple", "Orange" -> 52;

case "Banana" -> 89;

case "Strawberry" -> 32;

default -> throw new IllegalArgumentException("Unknown fruit: " + fruit);

};

System.out.println("Calories in " + fruit + ": " + calories);

}

}

**Example 3: Enhanced Switch with Block**

public class EnhancedSwitchExample3 {

public static void main(String[] args) {

String month = "February";

int days = switch (month) {

case "January", "March", "May", "July", "August", "October", "December" -> 31;

case "April", "June", "September", "November" -> 30;

case "February" -> {

System.out.println("Checking if leap year...");

boolean leapYear = false; // Simplified leap year check

yield leapYear ? 29 : 28;

}

default -> throw new IllegalArgumentException("Unknown month: " + month);

};

System.out.println("Days in " + month + ": " + days);

}

}

**Example 4: Switch Expression with Enum**

public class EnhancedSwitchExample4 {

enum TrafficLight {

RED, YELLOW, GREEN

}

public static void main(String[] args) {

TrafficLight signal = TrafficLight.RED;

String action = switch (signal) {

case RED -> "Stop";

case YELLOW -> "Caution";

case GREEN -> "Go";

};

System.out.println("Traffic light action: " + action);

}

}

**Example 5: Nested Enhanced Switch**

public class EnhancedSwitchExample5 {

public static void main(String[] args) {

int year = 2023;

int month = 2;

int day = 28;

String dayType = switch (month) {

case 1, 3, 5, 7, 8, 10, 12 -> "31-day month";

case 4, 6, 9, 11 -> "30-day month";

case 2 -> switch (day) {

case 29 -> "Leap year February";

default -> {

boolean leapYear = (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);

yield leapYear ? "Leap year February" : "Non-leap year February";

}

};

default -> "Invalid month";

};

System.out.println("The month and day type: " + dayType);

}

}

# OOPS Concepts

## Introduction to class

A **class** is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. This includes classes for objects occurring more than once in your code.

**Components of a Class**

In general, class declarations can include these components in order:

**A. Modifiers**

The class can be either public or have default access. If no modifier is specified, it has default access. It is also known as Access specifier or Visibility mode.

**Access Modifiers / Access Specifiers / Visibility Modes**

In Java, methods and data members can be encapsulated by the following four access modifiers. The access modifiers are listed according to their restrictiveness order:

1. **private**: Accessible within the class where defined.
2. **default** (package-private): When no access modifier is specified.
3. **protected**: Accessible only to classes that subclass your class directly within the current or different package.
4. **public**: Accessible from any class.

However, the classes and interfaces themselves can have only two access modifiers when declared outside any other class:

1. **public**
2. **default**: When no access modifier is specified.

**Note**:

* Nested interfaces and classes can have all access modifiers.
* We cannot declare class/interface with private or protected access modifiers.

**B. Class Name**

It should begin with the initial letter capitalized by convention. The name of the class would depend on the type of work performed by the program. For example, if we want to make a class with methods related to arithmetic calculations, we might name our class Calculations.

public class Main {

public static void main(String[] args) {

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

}

}

**C. Superclass (if any)**

If the class has a parent (superclass), its name is preceded by the keyword extends. A class can only extend (subclass) one parent. If a class doesn’t extend any class, then by default it extends the Object class (the parent class of all classes) implicitly.

// This is a parent class

class First {

public void display() {

System.out.println("This is parent class");

}

}

// This is a child class

public class Main extends First {

public static void main(String[] args) {

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

}

}

**Note**: A child class has the capability to access most of its parent class members except:

1. Constructors of the parent class
2. Static members of the parent class
3. Private members of the parent class
4. Static and non-static blocks of the parent class

**D. Interfaces (if any)**

If the class implements interfaces, their names are listed after the implements keyword, separated by commas. A class can implement more than one interface. In Java, an interface specifies the behavior of a class by providing an abstract type. Interfaces are used in Java to achieve abstraction.

interface A {

public void implementDisplayFunction();

}

public class Main implements A {

public void implementDisplayFunction() {

System.out.println("This method is successfully implemented");

}

public static void main(String[] args) {

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

}

}

**E. Body**

The class body, where you define fields, methods, constructors, etc., is surrounded by braces { }.

public class Main extends First {

public static void main(String[] args) {

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

}

}

**F. Methods**

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping it, which is why they are considered time savers. In Java, every method must be part of some class, which is different from languages like C, C++, and Python.

**Example Methods**

public class Main {

public int multiplication(int a, int b) {

return a \* b;

}

public int addition(int a, int b) {

return a + b;

}

public double average(int a, int b) {

return (a + b) / 2.0;

}

public static void main(String[] args) {

Main obj = new Main();

System.out.println("Multiplication: " + obj.multiplication(2, 3));

System.out.println("Addition: " + obj.addition(2, 3));

System.out.println("Average: " + obj.average(2, 3));

}

**Complete Example Program**

Here's a complete example that includes all the components discussed:

// A. Modifiers: public class

public class Main extends Parent implements A {

// B. Class Name: Main

// C. Superclass: Parent

// D. Interfaces: A

// E. Body: Class body starts here

// Method from interface A

public void implementDisplayFunction() {

System.out.println("This method is successfully implemented");

}

// Methods

public int multiplication(int a, int b) {

return a \* b;

}

public int addition(int a, int b) {

return a + b;

}

public double average(int a, int b) {

return (a + b) / 2.0;

}

// Main method

public static void main(String[] args) {

Main obj = new Main();

obj.display(); // From Parent class

obj.implementDisplayFunction(); // From interface A

System.out.println("Multiplication: " + obj.multiplication(2, 3));

System.out.println("Addition: " + obj.addition(2, 3));

System.out.println("Average: " + obj.average(2, 3));

}

// E. Body: Class body ends here

}

// Parent class

class Parent {

public void display() {

System.out.println("This is parent class");

}

}

// Interface A

interface A {

public void implementDisplayFunction();

}

This example includes a class Main that extends a parent class Parent and implements an interface A. It has methods for multiplication, addition, and average calculations, and demonstrates the use of these methods in the main method.

# HTML code Libraries

To format code in a box and highlight keywords for different programming languages, you can use various libraries and tools. Here’s a list of popular libraries and frameworks for these purposes, which can be used in different environments:

**Libraries for Code Formatting and Syntax Highlighting**

1. **PrismJS**
   * **Description**: A lightweight, extensible syntax highlighter for web pages.
   * **Features**: Supports a wide range of languages and provides themes for code highlighting.
   * **Usage**: Easily integrate into web pages using <script> and <link> tags.
   * **Website**: [PrismJS](https://prismjs.com/)

html

Copy code

<link href="https://cdn.jsdelivr.net/npm/prismjs@1.27.0/themes/prism.min.css" rel="stylesheet"/>

<script src="https://cdn.jsdelivr.net/npm/prismjs@1.27.0/prism.min.js"></script>

1. **Highlight.js**
   * **Description**: A library that provides syntax highlighting for over 180 languages.
   * **Features**: Auto-detects language, supports custom themes, and is very easy to set up.
   * **Usage**: Include via CDN or npm, and add <pre><code> tags to your HTML.
   * **Website**: [Highlight.js](https://highlightjs.org/)

html

Copy code

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/highlight.js/11.8.0/styles/default.min.css">

<script src="https://cdnjs.cloudflare.com/ajax/libs/highlight.js/11.8.0/highlight.min.js"></script>

<script>hljs.highlightAll();</script>

1. **Pygments**
   * **Description**: A Python library for syntax highlighting that can be used to generate highlighted code in various formats (HTML, LaTeX, etc.).
   * **Features**: Extensive language support, highly customizable, can be used from the command line or integrated into Python code.
   * **Usage**: Typically used for generating static HTML pages with highlighted code.
   * **Website**: [Pygments](https://pygments.org/)

bash

Copy code

pygmentize -o output.html input.py

1. **CodeMirror**
   * **Description**: A versatile text editor implemented in JavaScript for the web with syntax highlighting.
   * **Features**: Full-fledged editor with support for many languages and a rich set of features.
   * **Usage**: Include via CDN or npm, and initialize with JavaScript.
   * **Website**: [CodeMirror](https://codemirror.net/)

html

Copy code

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/codemirror/5.65.2/codemirror.min.css">

<script src="https://cdnjs.cloudflare.com/ajax/libs/codemirror/5.65.2/codemirror.min.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/codemirror/5.65.2/mode/javascript/javascript.min.js"></script>

1. **Ace Editor**
   * **Description**: A high-performance code editor written in JavaScript that runs in the browser.
   * **Features**: Syntax highlighting, code folding, and autocompletion for a variety of languages.
   * **Usage**: Include via CDN or npm, and initialize in your HTML or JavaScript.
   * **Website**: Ace Editor

html

Copy code

<script src="https://cdnjs.cloudflare.com/ajax/libs/ace/1.4.14/ace.min.js"></script>

<div id="editor">console.log("Hello, world!");</div>

<script>

var editor = ace.edit("editor");

editor.session.setMode("ace/mode/javascript");

</script>

**Example Usage in HTML**

Here’s a sample HTML file demonstrating how to use **PrismJS** for code formatting and syntax highlighting:

html

Copy code

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Code Highlighting Example</title>

<link href="https://cdn.jsdelivr.net/npm/prismjs@1.27.0/themes/prism.min.css" rel="stylesheet"/>

<style>

pre {

background: #f5f5f5;

padding: 10px;

border: 1px solid #ddd;

border-radius: 5px;

overflow: auto;

}

</style>

</head>

<body>

<h1>Code Highlighting Example with PrismJS</h1>

<pre><code class="language-java">

public class HelloWorld {

public static void main(String[] args) {

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

}

}

</code></pre>

<script src="https://cdn.jsdelivr.net/npm/prismjs@1.27.0/prism.min.js"></script>

</body>

</html>

**Explanation of the Code:**

* **PrismJS Integration**: Includes the PrismJS CSS and JavaScript files via CDN to handle syntax highlighting.
* **Code Block**: Wraps the code in <pre> and <code> tags with the appropriate language class (language-java in this case).
* **Styling**: Adds some basic styling to the <pre> tag to format the code block with a background and border.

This approach ensures that the code is displayed in a readable, highlighted format, enhancing the presentation and readability of code snippets.