Hello Chapter 1

General Concept of Programming

This syllabus provides a comprehensive breakdown of core and advanced Java concepts, along with resources for further exploration.

**I. Introduction to Java (1-2 weeks)**

* **History and Features of Java:**
  + Overview of Java's creation and evolution
  + Key features like platform independence, object-oriented nature, security
* **Setting up the Development Environment (IDE):**
  + Choosing an IDE (Integrated Development Environment) like Eclipse, IntelliJ IDEA
  + Installing and configuring the IDE
  + Understanding project creation, navigation, and debugging tools
* **Understanding Java Program Structure (1 week):**
  + Basic program structure (class, methods, main method)
  + Source files (.java) and compilation process
  + Running Java programs (command line vs IDE execution)

**Resources:**

* TutorialsPoint - Java Tutorial [invalid URL removed]
* Oracle Java Documentation - Getting Started [invalid URL removed]
* Online IDEs - JDoodle [invalid URL removed]

**II. Java Fundamentals (2-3 weeks)**

**A. Basic Syntax (1 week):**

* **Variables:**
  + Data types (primitive - int, double, char, etc.; reference - String, Object)
  + Declaring and initializing variables
* **Operators:**
  + Arithmetic operators (+, -, \*, /)
  + Logical operators (&&, ||, !)
  + Assignment operators (=, +=, -=)
  + Other operators (increment/decrement, relational, etc.)
* **Expressions and Statements:**
  + Combining operators and variables to form expressions
  + Using statements like if-else, for loops, to control program flow

**B. Control Flow (1 week):**

* **Conditional Statements:**
  + if-else statements for decision making
  + switch statements for multi-way branching
* **Looping Statements:**
  + for loops for repetitive execution with counter
  + while loops for indefinite execution based on condition
  + do-while loops for guaranteed execution at least once

**C. Functions (Methods) (1 week):**

* **Defining and Calling Methods:**
  + Creating reusable blocks of code
  + Passing arguments (parameters) to methods
* **Parameter Passing:**
  + By value (primitive data types) vs by reference (objects)
* **Method Overloading:**
  + Having multiple methods with the same name but different parameter lists
* **Recursion:**
  + A function calling itself

**Resources:**

* W3Schools - [Java Operators](https://www.w3schools.com/java/java_operators.asp)
* Javapoint - Java Control Flow Statements [invalid URL removed]
* GeeksforGeeks - [Java Methods](https://www.geeksforgeeks.org/methods-in-java/)

**III. Object-Oriented Programming (OOP) Concepts (2-3 weeks)**

**A. Classes and Objects (1 week):**

* **Defining Classes:**
  + Blueprints for creating objects
  + Specifying attributes (variables) and methods (functions)
* **Creating Objects:**
  + Instantiating classes to create objects
  + Accessing object members (attributes and methods)

**B. Inheritance (1 week):**

* **Subclasses and Superclasses:**
  + Creating hierarchical relationships between classes
  + Code reusability through inheritance
* **Polymorphism:**
  + Overriding methods in subclasses for different behavior
  + Method overriding vs overloading

**C. Abstraction (0.5-1 week):**

* **Abstract Classes and Interfaces:**
  + Defining abstract classes with unimplemented methods
  + Interfaces for specifying what a class must do (without implementation)

**D. Packages (0.5-1 week):**

* **Organizing Classes:**
  + Grouping related classes into packages for better structure and reusability

**Resources:**

* Oracle Java Documentation - Object-Oriented Programming [invalid URL removed]
* Baeldung - [Java Inheritance](https://www.baeldung.com/java-inheritance)
* Programiz - Java Abstraction [invalid URL removed]

**IV. Advanced Java Topics (3-4 weeks)**

**A. Exception Handling (1 week):**

* **Try-Catch Blocks:**
  + Handling errors and exceptions that occur during program execution

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

# Introduction to programming

Programming, at its core, is the process of giving a computer a set of instructions that it can understand and execute. These instructions, called code, are written in a specific language that the computer can interpret. Imagine it as giving a detailed recipe to a kitchen appliance that can follow every step precisely.

**Languages of Instruction**

* Just like humans have different languages, computers have their own languages for programming. These languages are designed with specific syntax (rules for structuring the code) and semantics (meaning attached to the code).
* Popular programming languages include Python, Java, C++, JavaScript, and many more. Each language has its strengths and weaknesses, making it suitable for different tasks.

Giving Instructions (Coding)

* Programming involves writing code that tells the computer what to do step-by-step. This code can be as simple as performing calculations or as complex as building entire applications.
* The code typically involves:
  + **Variables**: Storing data like numbers, text, or even collections of data.
  + **Operators**: Performing operations on data (addition, comparison, etc.).
  + **Control Flow Statements**: Deciding which instructions to execute based on conditions (if-else statements, loops).
  + **Functions/Methods**: Reusable blocks of code that perform specific tasks.

Translating to Machine Code

* While programmers write code in a human-readable language, computers can only understand machine code, a series of **0s** and **1s**.
* A translator called a compiler or interpreter converts the programmer's code into machine code that the computer can execute.

Problem-Solving and Logic

* Programming is a powerful tool for solving problems. It requires breaking down complex tasks into smaller, logical steps that the computer can understand.
* Programmers need to think critically, analyze problems, and design algorithms (step-by-step solutions) that can be translated into code.

Beyond the Basics

* As you progress in programming, you'll encounter more advanced concepts like object-oriented programming (OOP), data structures, algorithms, and software development methodologies.
* OOP helps create reusable and modular code, while data structures organize information efficiently, and algorithms provide efficient solutions to problems. Software development methodologies establish best practices for building large software systems.

Examples of Programming in Action

* The websites you visit, the apps you use on your phone, the games you play - all rely on programming to function.
* Programmers create software for various purposes, including:
  + Developing web applications
  + Building mobile apps
  + Creating video games
  + Automating tasks
  + Data analysis and machine learning

# **Programming Paradigm**

Paradigm can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach. There are lots for programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming language there are lots of paradigms to fulfill each and every demand.

**Types of programming paradigm**

1. Imperative Programming Paradigm
2. Declarative Programming Paradigm
3. Object-Oriented programming paradigm
4. Functional programming Paradigm
5. Logic Programming Paradigm
6. Reactive Programming Paradigm

note: Here we explained most popular programming paradigm only.

1. Imperative Programming Paradigm

Imagine you're giving instructions to a friend on how to bake a cake. Imperative programming works similarly. It's all about giving the computer a set of explicit instructions, step-by-step, to achieve a specific outcome.

**Key Concepts**

* **Focus on State** Programs manipulate data (variables) and change their state (values) throughout execution.
* **Control Flow Statements** if-else, for, while statements are used to control the flow of execution based on conditions and loops.
* **Procedures/Functions** Reusable blocks of code that perform specific tasks, often taking input and returning output.

**Python Example**

def bake\_cake(ingredients, temperature, time):

# Mix ingredients (changing state of ingredients variable)

mixed\_batter = mix(ingredients)

# Put batter in oven (changing state of batter location)

oven.bake(mixed\_batter, temperature, time)

# Take cake out (changing state of cake location)

return cake

cake = bake\_cake(["flour", "sugar", "eggs"], 350, 30)

**Strengths**

* Simple and intuitive for beginners, easy to reason about program flow.
* Efficient for low-level tasks and hardware manipulation.

**Weaknesses**

* Code can become verbose and difficult to maintain for complex problems.
* Error handling can be cumbersome.

1. Declarative Programming Paradigm

Instead of giving step-by-step instructions, declarative programming focuses on what you want the program to achieve. It's like describing the cake you desire, and the computer figures out the recipe and baking process.

**Key Concepts**

* **Focus on What** Programs specify the desired outcome or goal without detailing the "how."
* **Declarative Statements** High-level statements define the properties or relationships between data.
* **Less Control Flow** Focuses on the result rather than the specific execution steps.

SQL Example

SELECT \* FROM cakes WHERE temperature > 300 AND time > 25;

This query declares the desired outcome (selecting cakes) based on specific conditions (temperature and time) without specifying how to find them in the database.

**Strengths**

* Code can be concise and easier to maintain for complex problems.
* Focus on desired outcome improves readability.

**Weaknesses**

* May be less intuitive for beginners, requires a different way of thinking about problems.
* Limited control over specific execution steps.

1. Object-Oriented Programming (OOP)

OOP takes inspiration from the real world. It organizes code around objects, which are self-contained entities that encapsulate data (attributes) and related behaviors (methods). Imagine a cake object that has attributes like ingredients, temperature, and methods like bake() and frost().

**Key Concepts**

* **Objects:** Combine data and functionality, representing real-world entities.
* **Classes:** Blueprints for creating objects, defining attributes and methods.
* **Inheritance:** Allows creating new classes (subclasses) that inherit properties and behaviors from existing classes (superclasses).
* **Encapsulation:** Protects data integrity by restricting direct access to attributes.

**Example (Java - OOP):**

Java

class Cake {

private String ingredients; // Encapsulated attribute

private int temperature;

public void bake(int time) {

// Baking logic using ingredients and temperature

}

}

class ChocolateCake extends Cake { // Inheritance

public void frost() {

// Frosting logic specific to chocolate cake

}

}

**Strengths:**

* Promotes modularity, reusability, and code maintainability.
* Makes code more organized and easier to understand for complex systems.

**Weaknesses:**

* Can be more complex to learn and design compared to imperative programming.
* Might be overkill for simple tasks.

1. Functional Programming:

Here, programs are built around the concept of pure functions. These functions take input and always return the same output for that given input, without causing side effects (changes to global state). It's like following a precise recipe every time to ensure consistent cake results.

**Key Concepts:**

* **Pure Functions:** No side effects, always return the same output for a given input.
* **Immutability:** Data is treated as immutable (unchanging) after creation, leading to new data structures for modifications.
* **Recursion:** Functions can call themselves, breaking down problems into smaller subproblems until a base case is reached.

**Example (Haskell - Functional):**

mixIngredients :: [String] -> [String]

mixIngredients ingredients = concat [map toUpper ingredient | ingredient <- ingredients]

bakeCake :: Int -> Int -> [String] -> [String]

bakeCake temperature time ingredients = mixIngredients ingredients ++ ["Baked for", show time, "minutes at", show temperature, "degrees"]

This code defines pure functions for mixing ingredients and baking a cake, avoiding side effects and emphasizing immutability.

**Strengths:**

* Encourages code that is predictable, easier to test and reason about.
* Well-suited for parallel processing and concurrency.

**Weaknesses:**

* Can be less intuitive for beginners due to its emphasis on immutability and recursion.
* May not be the most efficient choice for all types of problems.

Choosing the Right Paradigm:

The best paradigm for a project depends on the specific problem you're trying to solve. Here are some general guidelines:

* **Imperative:** Well-suited for low-level tasks, hardware interaction, and tasks requiring precise control flow.
* **Declarative:** Ideal for complex data manipulation, configuration management, and querying databases.
* **OOP:** Effective for modeling real-world entities, building large-scale applications, and promoting code reusability.
* **Functional:** Excellent for data analysis, parallel processing, and creating predictable and reliable code.
* **Other Paradigms:** Apply to specific problem domains and can be used in conjunction with other paradigms.

# **Java Introduction**

Java is a programming language and computing platform first released by Sun Microsystems in 1995. It has evolved from humble beginnings to power a large share of today’s digital world, by providing the reliable platform upon which many services and applications are built. New, innovative products and digital services designed for the future continue to rely on Java, as well.

While most modern Java applications combine the Java runtime and application together, there are still many applications and even some websites that will not function unless you have a desktop Java installed. Java.com, this website, is intended for consumers who may still require Java for their desktop applications – specifically applications targeting Java 8. Developers as well as users that would like to learn Java programming should visit the [dev.java](https://dev.java/) website instead and business users should visit [oracle.com/java](https://www.oracle.com/java/) for more information.

**Inventor Of Java**

A person with a beard and glasses

Description automatically generatedJames Gosling, a Canadian computer scientist, invented Java. In 1991, while working at Sun Microsystems, he started the Java project. The project's initial goal was to create a platform-independent language for interactive television, but the digital cable television industry at the time was not yet ready for it.

Gosling and his team created a new virtual machine and a new programming language, which they called Java. Java was designed to be portable, secure, and easy to use. It quickly became popular for developing web applications, and it is now one of the most widely used programming languages in the world.

Gosling has received numerous awards for his work on Java, including the ACM Software Systems Award and the IEEE John von Neumann Medal. He is a member of the National Academy of Engineering and the Royal Society of Canada.

**1. The Birth of Java (1991-1995):**

1. **Green Team Origins:** Created by James Gosling, Bill Joy, Mike Sheridan, and Guy Steele at Sun Microsystems (later acquired by Oracle).
2. Project "**Oak**": The initial name, inspired by an oak tree outside Gosling's office.
3. Focus on Embedded Systems: Designed for consumer electronics like set-top boxes, targeting real-time performance and a small memory footprint.

**2. The Rise of the Web (1995-2000):**

1. **A Web-Ready Language**: Netscape's interest in developing interactive web content led to Java's adaptation for the web.
2. **Applets and Security**: Applets, small Java programs embedded in web pages, provided dynamic content. The Java Virtual Machine (JVM) ensured platform independence and security by sandboxing applets.
3. **Java 1.0 Released (1996)**: The official public release marked the beginning of widespread adoption.

**3. Enterprise Boom and Beyond (2000-2010):**

1. **J2EE (Java 2 Platform, Enterprise Edition**): A suite of APIs and specifications for building large-scale enterprise applications.
2. **Open Source Movement**: The release of the Java Development Kit (JDK) as open-source software in 2006 fostered community development and innovation.
3. **Mobile Java (J2ME):** An attempt to establish Java in the mobile market, facing competition from native development tools.

**4. The Modern Java Landscape (2010-Present):**

1. **Rise of Android (2008):** Java became the primary programming language for Android app development, significantly boosting its popularity in mobile computing.
2. **Java 7 and 8 (2011-2014):** Introduced significant improvements like closures, lambda expressions, and the Fork/Join framework for parallel processing.
3. **Focus on Cloud and Microservices**: Modern Java frameworks like Spring Boot and technologies like Docker containers cater to cloud-based development and microservices architectures.
4. **Java 11 and Beyond (2018-Present):** Modularization, improved garbage collection, and features like long support releases (LTS) enhance developer experience and long-term application stability.
5. **Java as Programming Language**

Java is multi paradigm programming language. It is one of the most used programming

language for development of various types of software such as desktop, enterprise, web

based and mobile applications.

* 1. **Statically typed** – The type of the variable is known at compile time.
  2. **Object oriented** – Object centered programming.
  3. **Concurrent** – Support for multithreading programming.
  4. **Reflective** – Allows inspection of class, method, interface, fields.

**Why use JAVA**

1. Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.).
2. It is one of the most popular programming languages in the world.
3. It has a large demand in the current job market.
4. It is easy to learn and simple to use.
5. It is open-source and free.
6. It is secure, fast and powerful.
7. It has huge community support (tens of millions of developers).
8. Java is an object-oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs.
9. As Java is close to [C++](https://www.w3schools.com/cpp/default.asp) and [C#](https://www.w3schools.com/cs/default.asp), it makes it easy for programmers to switch to Java or vice versa.

# **Features of Java**

Java's success stems from its well-designed features that promote code quality, efficiency, and maintainability. Here's a breakdown of key Java features:

**Basic**

1. **Simple**

* A hand pointing at a sign

  Description automatically generatedEasy to learn and use, especially for those familiar with C++ or other similar languages.
* Java eliminates complex features like pointers, operator overloading, etc., making it simpler and less error prone.
* Its syntax is clear and straightforward, promoting readability and maintainability of code.
* Provides automatic memory management through garbage collection, reducing the burden on developers to manage memory manually.

1. **Object-Oriented Programming (OOP)**
   * A blue and white computer screen with green and white text

     Description automatically generatedJava is an object-oriented language, meaning programs are built around objects.
   * An object encapsulates data (attributes) and related behaviors (methods).
   * OOP principles like inheritance, polymorphism, and encapsulation promote code reusability, modularity, and maintainability.
2. A black and white toolbox

   Description automatically generated**Portable**

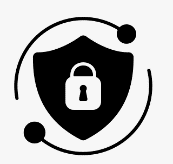
* Java programs are compiled into bytecode, which can be executed on any platform that has a Java Virtual Machine (JVM) installed.
* The "Write Once, Run Anywhere" (WORA) principle allows Java applications to run on various devices without modification.
* JVM implementations are available for different operating systems, ensuring portability across diverse environments.
* Abstracts away hardware specific functionalities, enhancing platform independence.

1. **Platform Independence ("Write Once, Run Anywhere" - WORA)**

* A black and white computer with a gear

  Description automatically generatedJava code is compiled into bytecode, an intermediate representation that can run on any platform with a Java Virtual Machine (JVM).
* The JVM interprets the bytecode and translates it into machine code specific to the underlying system.
* This feature allows developers to write code once and deploy it across various operating systems (Windows, Linux, macOS, etc.) without modifications.

1. **Secure**

* Java's design prioritizes security by incorporating features like:
* Absence of explicit pointers (reducing vulnerabilities like buffer overflows)
* Secure runtime environment (JVM sandboxing)
* Strong typing (reducing type conversion errors)
* These features make Java a popular choice for developing secure applications.

1. **Robust**

* A black chain link

  Description automatically generatedStrong memory management and exception handling mechanisms ensure robustness against errors and crashes.
* Type safety features prevent common programming errors like null pointer exceptions, array out-of-bounds errors, etc.
* Java's object-oriented approach promotes modularity and encapsulation, leading to more resilient and maintainable code.
* Strict compile time and runtime checking help detect and eliminate errors early in the development process.

1. **Architecture Neutral**

* A black and white computer chip

  Description automatically generatedJava's bytecode is platform independent, allowing Java applications to run on any device with a compatible JVM.
* Decouples the development process from the underlying hardware architecture, enabling seamless deployment across different systems.
* JVM implementations abstract away low-level details, providing a uniform execution environment regardless of the underlying hardware.
* Enhances scalability and interoperability by ensuring compatibility with diverse architectures and devices.

1. **Interpreted**

* A person with a headset and a globe

  Description automatically generatedJava programs are initially compiled into bytecode, which is then interpreted by the JVM at runtime.
* This interpretation process enables platform independence, as the same bytecode can be executed on any system with a compatible JVM.
* Interpreted execution allows for dynamic adaptation to runtime conditions, optimizing performance and resource utilization.
* Just In time (JIT) compilation techniques further improve performance by dynamically translating bytecode into native machine code during execution.

1. **High Performance**

* A black gauge with a point

  Description automatically generatedJava's Just in Time (JIT) compilation optimizes performance by translating bytecode into native machine code at runtime.
* Advanced memory management techniques, such as garbage collection, minimize memory leaks and optimize resource utilization.
* Multithreading support enables concurrent execution of tasks, leveraging the full potential of modern multicore processors.
* Java's extensive standard library and ecosystem of third-party frameworks provide efficient solutions for various tasks, enhancing overall performance.

1. A black and white cloud with gears

   Description automatically generated**Multithreading**

* Java allows creating multiple threads of execution within a program.
* This enables concurrent processing, improving performance for tasks that can benefit from parallelism.

1. **Distributed**

* A black and white logo of a truck with a box and pin

  Description automatically generatedBuilt-in networking capabilities allow Java applications to communicate over a network, facilitating distributed computing.
* Java's Remote Method Invocation (RMI) framework enables seamless interaction between distributed objects.
* Support for industry standard protocols like HTTP, TCP/IP, and SOAP simplifies the development of distributed applications.
* Java Enterprise Edition (Java EE) provides robust APIs and frameworks for building scalable and resilient distributed systems.

1. **Dynamic**

* A black and white logo

  Description automatically generatedJava supports dynamic loading of classes and resources at runtime, enabling flexible and extensible applications.
* Reflection APIs allow Java programs to inspect and manipulate objects dynamically, enhancing runtime flexibility.
* Dynamic method invocation and polymorphism enable adaptive behavior based on runtime conditions.
* Java's modular system (introduced in Java 9) facilitates dynamic assembly and deployment of modular applications.

**Extra**

1. **Garbage Collection**

* A black background with a black square

  Description automatically generated with medium confidenceJava includes automatic garbage collection that manages memory allocation and deallocation.
* This feature reduces the risk of memory leaks and simplifies memory management for developers.

1. **Strong Typing**

* A black and white illustration of hands on a keyboard

  Description automatically generatedJava is a statically typed language, meaning variables must be declared with a specific data type (e.g., int, double, String).
* This helps catch potential errors during compilation, improving code reliability.

1. A black and white icon of a document

   Description automatically generated**Exception Handling**

* Java provides mechanisms for handling errors and exceptions that may occur during program execution.
* Using try-catch blocks, developers can gracefully handle exceptions and prevent program crashes.

1. A black and white image of a stack of books

   Description automatically generated**Rich Set of Libraries and APIs (Application Programming Interfaces)**

* Java provides a vast library of pre-written code for various functionalities.
* The Java API offers classes for common tasks like networking, file I/O, GUI development, and database interactions.
* Developers can leverage these libraries to streamline development and avoid reinventing the wheel.

1. **Rich Development Tools and IDEs (Integrated Development Environments)**

* A computer screen with a computer code

  Description automatically generatedA wide range of development tools and IDEs are available for Java, providing features like code completion, syntax highlighting, debugging, and project management.
* These tools enhance developer productivity and improve the overall development experience.

1. A black and white symbol with circles and people

   Description automatically generated**Large and Active Community**

* Java boasts a large and active community of developers worldwide.
* This community provides extensive online resources, tutorials, forums, and libraries, making it easier for beginners to learn and experienced developers to find solutions.

**BEYOND THE BASICS**

As you delve deeper into Java, you'll encounter even more advanced features:

* **Annotations**: Provide metadata for code, used for reflection and configuration.
* **Generics**: Enable creating type-safe collections and methods that can work with various data types.
* **Lambda Expressions (Java 8+):** Provide concise syntax for defining anonymous functions, promoting functional programming style.
* **Java Modules (Java 9+):** Modularize large codebases, improving maintainability and reducing classpath conflicts.

# Why Java is secure

1. **JVM**

JVM is Java Virtual Machine. Its basic role is to verify the bytecode before it is run. This makes sure that the program isn't making any unsafe operations. There are various unsafe operations that the programs can generally perform; for example, a program might branch to incorrect locations that can contain data rather than instruction. The JVM ensures that such unsafe operations aren't being performed. Over and above this task, the JVM also enforces runtime constraints. This can include array bounds checking and others like it reduces the chances of the developers suffering from memory safety flaws. Hence, they can avoid such flaws as buffer overflow or others, which makes java more secured language as compared to other languages

1. **Security Manager**

Security manager layer is present in JVM. It makes sure that the untrusted code doesn't manage to access some APIs and features of the platform. Java does not support pointers (C & C++ supports).

1. **No Pointers**

Some of the arbitrary memory locations can be addressed with the help of pointers for doing read and write operations which are unauthorized. This does not serve the purpose of being secured. That is why users do not use the concept of pointers.

1. **Access Modifiers**

There are four access modifiers in java i.e. public, protected, private and default. So as developer if we want to hide some code then we can use access modifiers.

If we want that our code or data should not be override, then we can use 'final' keyword; which again adds security to java.

1. **Exception Handling**

Exception handling catch the results which are unexpected through exception handling and report the error to the programmer. Until the rectification of the error by the programmer this concept will not allow us to run the code. Thereby proving Java's security.

1. **Own Memory Management**

The memory management mechanism is unique and is owned by Java. There is no need for manual intervention for garbage collection and everything is handled automatically.

There is no need for a headache to free the memories. It drastically reduces the programmer overhead. Therefore the programmer's hand must be free from memory management. Relieving the memory in Java is the job of JVM.

# Uses Of Java

1. **Android Apps**

|  |
| --- |
| Java has a rich use in Android Applications. Open your Android phone and any app, they are actually written in Java programming language, with Google's Android API, which is similar to JDK (DVM in android). Couple of years back Android has provided much needed boost and today many Java programmer are Android App developer. |

|  |  |
| --- | --- |
| 1. **Financial Services Industries**   Java is very big in Financial Services which demands more security. API's of Data Processing & Payment Gateways are created in java. It is mostly used to write server side application, mostly without any front end, which receives data form one server (upstream), process it and sends it other process (downstream). | |
| 1. **Big Data**   Many social networking websites like facebook twitter etc. Technologies : and e-commerse websites generates a lot of data day to day. So to manage this huge amount of data, Hadoop and other big data technologies are developed using java only. So Java plays an important role for big data technologies. | |

|  |
| --- |
| 1. **Scientific**   As Java is more safe, portable, maintainable and comes Applications : with better high-level concurrency tools than C++ or any other language therefore Java is often a default choice for scientific applications, including natural language processing. |

|  |
| --- |
| 1. **Financial Services Industries**   Java is very big in Financial Services which demands more security. API's of Data Processing & Payment Gateways are created in java. It is mostly used to write server side application, mostly without any front end, which receives data form one server (upstream), process it and sends it other process (downstream). |

|  |
| --- |
| 1. **Java Web**   Java is also big on E-commerce and web application space. Applications : Java frameworks i.e. Spring MVC, Struts 2.0 and other frameworks are used for this web applications. Many of government, healthcare, insurance, education, defense and several other department have their web application built in Java. |

|  |
| --- |
| 1. **Software tools**   Many useful software and development tools are written and developed in Java e.g. Netbans, Eclipse and InetelliJ IDE. They are also most used desktop applications for development of java and other languages. |

|  |
| --- |
| 1. **Big Data Technologies**   Many social networking websites like facebook twitter etc and e-commerse websites generates a lot of data day to day. So to manage this huge amount of data, Hadoop and other big data technologies are developed using java only. So Java plays an important role for big data technologies. |

|  |
| --- |
| 1. **Scientific Applications**   As Java is more safe, portable, maintainable and comes with better high-level concurrency tools than C++ or any other language therefore Java is often a default choice for scientific applications, including natural language processing. |

|  |
| --- |
| 1. **Embedded Systems**   Java is also used in embedded systems. Computers, Printers, Routers, ATM's, Home Security Systems etc. all uses java a lot. |

|  |
| --- |
| 1. **J2ME Apps**   There was time when Nokia and Samsung handsets had a large market which uses **J2ME**. At that time almost all games, applications, which is available in Android are written using **MIDP** and **CLDC**, part of **J2ME** platform. |

# **Modern programming vs Non-Modern Programming**

The world of programming is constantly evolving, with new languages, frameworks, and practices emerging all the time. Here's a breakdown of the key differences between modern and non-modern programming approaches:

1. **Paradigms and Languages**

**Modern:** Leans towards high-level languages that focus on readability, maintainability, and developer productivity. Examples include Python, Java, JavaScript (with frameworks), C#, and Go. These languages often embrace paradigms like object-oriented programming (OOP) and functional programming, promoting code reusability and modularity.

**Non-Modern:** Often relies on lower-level languages closer to the machine's instruction set, like C, C++, and Assembly. These languages offer more control over hardware resources but require a deeper understanding of computer architecture and can be more error-prone.

1. **Development Tools and Practices**

**Modern**: Emphasizes integrated development environments (IDEs) that provide features like code completion, syntax highlighting, debugging tools, and version control integration. This makes development faster and less error-prone. Version control systems like Git allow for collaboration and easy tracking of code changes. Unit testing frameworks are widely used to ensure code quality and functionality.

**Non-Modern**: May use simpler text editors for code creation. Debuggers might be more basic, and version control might be manual or involve cumbersome systems. Unit testing might be less emphasized.

1. **Focus and Applications**

**Modern**: Often targets higher-level functionalities like web development, mobile app development, data science, machine learning, and cloud computing. Frameworks and libraries provide pre-built components and functionalities, allowing developers to focus on core logic.

**Non-Modern:** Concentrates more on system programming, embedded systems, device drivers, and performance-critical applications. Developers have more control over hardware interactions but need to handle low-level details.

1. **Security**

**Modern**: Languages often have built-in features for memory management and type safety, reducing the risk of security vulnerabilities like buffer overflows that can be common in non-modern languages.

**Non-Modern:** Security is more dependent on the programmer's expertise in handling memory allocation and data types.

It's important to note that these are generalizations. There's still a place for non-modern languages in specific scenarios where performance or hardware control is critical. Modern languages are constantly incorporating features and paradigms from non-modern approaches for better optimization when needed.

1. **Here's an analogy**

Imagine building a house. Non-modern programming is like using bricks and mortar directly, giving you fine-grained control but requiring a lot of effort. Modern programming provides pre-fabricated walls and pre-built plumbing systems, making construction faster and easier while still allowing customization.

# **Role Of Java in Front – End and Back – End**

**Back-End Development (Express / Heavy Role)**

Java shines in back-end development, where it excels at creating robust, scalable, and secure server-side applications. Here's a breakdown of the key technologies:

1. **Java Platform, Standard Edition (Java SE):** The foundation for back-end development with Java. It provides core libraries for I/O operations, networking, database interaction, multithreading, and more.
2. **Java Servlet API:** A cornerstone of back-end Java development, enabling creation of web servlets that handle HTTP requests and responses. Servlets interact with databases, session management, and other back-end services.
3. **JavaServer Pages (JSP):** A templating language that simplifies servlet development by combining HTML, Java code snippets (called scriptlets), and expressions within a single file. This allows for dynamic content generation based on user input or server-side data.
4. **Java Frameworks:** Streamline back-end development by providing pre-built components and patterns. Popular choices include:
5. **Spring Framework:** A comprehensive, modular framework for building enterprise-grade applications, offering features like dependency injection, security, and data access.
6. **Jakarta EE (formerly Java EE):** A collection of specifications for developing web services, enterprise applications, and distributed systems. Includes technologies like Java Persistence API (JPA) for database interaction, Java API for RESTful Web Services (JAX-RS) for building REST APIs, and Java Message Service (JMS) for asynchronous communication.
7. **Databases:** Relational (MySQL, PostgreSQL, Oracle) and NoSQL (MongoDB, Cassandra) databases can be accessed from Java using libraries like JDBC (Java Database Connectivity).
8. **Message Queues:** Frameworks like Apache ActiveMQ or RabbitMQ enable asynchronous communication between applications.
9. **Cloud Platforms:** Java applications can be deployed on cloud platforms like AWS, Azure, or GCP, leveraging their scalability, security, and other services.

A computer screen with a blue and pink image

Description automatically generated with medium confidence**Front-End Development (Limited Role)**

While Java isn't traditionally considered a front-end language, there are some niche use cases where it can be involved:

1. **Java Applets (Deprecated):** In the past, Java applets could be embedded in web pages to provide interactive elements. However, security concerns and the rise of more mature front-end technologies like JavaScript have led to their decline.
2. **GWT (Google Web Toolkit):** A framework that allows developers to write front-end code in Java and generate JavaScript code for better performance or leveraging existing Java skills. While GWT is still usable, it's not as widely adopted due to the popularity of modern JavaScript frameworks.

**Key Considerations**

1. **Front-End vs. Back-End Focus:** Java's strength lies primarily in back-end development. If your primary focus is front-end (user interface, interactivity), languages like JavaScript (with frameworks like React, Angular, or Vue.js) are more suitable.
2. **Full-Stack Development:** Java can be part of a full-stack developer's toolset, allowing them to work on both back-end and some niche front-end use cases.
3. **Project Requirements:** The choice of technologies ultimately depends on your project's specific requirements. Java remains a solid choice for robust back-end development, while other languages and frameworks might be more appropriate for front-end tasks.

*In conclusion, Java is a powerful back-end development language, and while it has some limited front-end applications, it's generally not the primary choice for front-end work. Understanding these distinctions will guide you in selecting the most suitable tools for your project.*

# Smart Abilities of Java

1. **Safety And Design in Java**

Java prioritizes safety in its **design**, **aiming** to protect against various threats such as **flawed code**, **viruses**, and **Trojan** horses. It achieves this through multiple layers of protection, including the Java virtual machine architecture and the class loader mechanism. These features establish a secure environment for running Java applications, enabling high-level security policies to control activities on an application basis.

Beyond security features, Java addresses common design and programming issues, making it safe from both human errors and legacy software problems. It emphasizes simplicity and provides useful tools, allowing users to build complex functionalities as needed.

|  |  |
| --- | --- |
| * **"Safety"** refers to the measures taken by Java to protect against various threats, including flawed code, viruses, and Trojan horses. * This includes features such as the Java virtual machine architecture and the class loader mechanism, which establish a secure environment for running Java applications. | * "**Design**" in this context encompasses the overall structure and approach of Java programming language. * It emphasizes simplicity and provides tools that address common design and programming issues, making Java safer from both human errors and problems inherited from legacy software. * The design philosophy of Java aims to keep the language simple while allowing users to build more complex functionalities on top of it when needed. |
|  |  |

1. **Language Simplicity**

In Java, simplicity is a guiding principle that influences various aspects of the language's design. Unlike some other languages, Java avoids features that tend to complicate code or lead to ambiguity. For instance, Java does not allow programmer-defined **operator overloading**, **source code preprocessing**, or **conditional compilation**. These features, often used for platform dependencies or debugging in other languages, are deemed unnecessary *in Java due to its sophisticated runtime optimizations and alternative solutions like assertions*.

* Additionally, Java provides a well-defined package structure for organizing class files, which streamlines code organization and reduces the need for extraneous context. Unlike languages like C/C++, Java does not require separate source "header" files, making Java code more readable and self-contained.
* Java also addresses structural issues differently. It supports single inheritance class hierarchy but allows multiple inheritance of interfaces. Interfaces, akin to abstract classes in other languages, specify object behavior without defining implementation. This powerful mechanism eliminates the need for multiple inheritance of classes, simplifying code structure and avoiding associated problems.
* Overall, Java's emphasis on simplicity results in a language that is clean, straightforward, and easier to understand, facilitating efficient development and maintenance of software projects.

1. **Type Safety in Java**

* Statically typed and dynamically typed languages, using examples like C/C++ and Smalltalk/Lisp, respectively. Statically typed languages define data types at compile time, enhancing error detection and enabling faster execution, but they lack flexibility. In contrast, dynamically typed languages perform type checking at runtime, allowing for more complex behavior but often at the expense of speed and safety.
* It delves into the concept of binding method calls to their definitions, contrasting early binding (compile-time binding) in languages like C/C++ with late binding (runtime binding) in languages like Smalltalk. Late binding offers flexibility but may incur runtime overhead.
* Java combines elements of both C++ and Smalltalk; it's statically typed yet late-binding. Objects in Java have well-defined types known at compile time, allowing for static type checking and usage analysis. However, Java's runtime system enables dynamic type inspection and supports late binding, facilitating flexibility like dynamically loaded objects and method overriding at runtime.

1. **Dynamic Memory Management**

* Dynamic memory management refers to the allocation and deallocation of memory during the execution of a program. In languages like C and C++, developers are responsible for manually allocating and deallocating memory using functions like malloc() and free(), which can lead to memory leaks and segmentation faults if not managed properly.
* However, in languages like Java, memory management is handled automatically by the JVM (Java Virtual Machine) through a process called garbage collection. Garbage collection automatically identifies and reclaims memory that is no longer in use, thus preventing memory leaks and simplifying memory management for developers.
* Java diverges significantly from lower-level languages like C and C++ in its memory management approach. It eliminates direct memory manipulation through pointers and introduces features like object garbage collection and high-level arrays, which enhance safety, portability, and optimization.

1. **Error Handling**

* Java, originally designed for networked devices and embedded systems, emphasizes robust error management. It features a powerful exception-handling mechanism like that of C++, providing a natural and elegant approach to error handling. Exceptions facilitate the separation of error-handling code from normal code, enhancing the readability and maintainability of applications.
* In Java, when an exception occurs, program execution is diverted to a designated "**catch**" **block**, which handles the exception. Each method in Java must either declare the exceptions it can generate or handle them within its body. This requirement ensures that error information is given the same importance as method arguments and return types, promoting the writing of correct and error-resilient software. As a Java programmer, you have clarity on the exceptional conditions to address, with assistance from the compiler in ensuring comprehensive error handling.
* Error handling is the process of identifying, responding to, and resolving errors that occur during the execution of a program. In Java, errors and exceptions are represented as objects, allowing developers to handle them using try-catch blocks. This enables graceful recovery from unexpected situations and improves the robustness of Java applications. Additionally, Java provides a mechanism for defining and throwing custom exceptions, allowing developers to create more informative error messages and streamline error handling logic.

1. **Threads**

* Modern applications demand parallelism for efficient multitasking, even in seemingly single-minded tasks with complex user interfaces. Threads in Java facilitate multiprocessing and task distribution for both client and server applications, simplifying their implementation due to built-in language support.
* While concurrency is valuable, effective thread programming requires synchronization to coordinate actions, which can be challenging without explicit language features. Java addresses this through synchronization mechanisms based on the monitor and condition model, ensuring safe resource access. The synchronized keyword designates methods and code blocks for serialized access within objects, supplemented by primitive methods for thread communication.
* Moreover, Java offers a high-level concurrency package, furnishing robust utilities for common multithreading patterns like thread pools and task coordination. With these features, Java boasts advanced thread-related functionalities compared to other languages. Despite the possibility of avoiding multithreaded code, mastering thread programming is essential in Java, enriching developers' skills and understanding of the language's capabilities.
* Threads are the smallest unit of execution within a process. In Java, multithreading allows concurrent execution of multiple threads within the same process, enabling developers to perform tasks concurrently and improve the responsiveness and performance of applications. Java provides built-in support for multithreading through the **Thread** **class** and the **Runnable** **interface**. Additionally, Java offers high-level concurrency utilities such as the Executor framework and the **java.util.concurrent** package, which simplify the development of multithreaded applications and ensure thread safety.

1. **Scalability**

Scalability refers to the ability of a system to handle increasing workload without sacrificing performance or responsiveness. In Java, scalability is achieved through various mechanisms such as multithreading, distributed computing, and load balancing. Java's platform independence and extensive library support make it well-suited for building scalable applications that can run on diverse hardware and operating systems. Additionally, Java's modular architecture and support for microservices enable developers to design and deploy scalable systems that can adapt to changing requirements and handle large volumes of traffic efficiently.

1. **Garbage Collector**

* The automatic garbage collector in Java is responsible for reclaiming memory that is no longer in use by the program. It periodically scans the **heap**, **identifying** and **deallocating** **objects** that are no longer reachable by the program. This process helps prevent *memory leaks* and *ensures efficient memory utilization*, freeing developers from the burden of manually managing memory allocation and deallocation. Java's garbage collector employs various algorithms such as the mark-and-sweep algorithm and the generational garbage collection to optimize performance and minimize pause times during garbage collection cycles.
* Garbage collection in Java automatically deallocates memory when objects are no longer in use, sparing programmers from the error-prone task of manual memory management. Java uses a sophisticated garbage collector that runs in the background, which means that most garbage collecting takes place during idle times, between I/O pauses, mouse clicks, or keyboard hits. Advanced runtime systems, such as Hotspot, have more advanced garbage collection that can differentiate the usage patterns of objects (such as short-lived versus long-lived) and optimize their collection. The Java runtime can now tune itself automatically for the optimal distribution of memory for distinct kinds of applications based on their behavior. With this kind of runtime profiling, automatic memory management can be much faster than the most diligently programmer-managed resources, something that some old-school programmers still find hard to believe.
* While Java does not have pointers in the traditional sense, it employs references, which are safer and strongly typed handles for objects. References in Java cannot be manipulated like pointers, preventing unauthorized memory access and enhancing security. Additionally, arrays in Java are first-class objects with built-in size and type information, reducing the need for pointer arithmetic and streamlining memory management.

# Compile time Vs Runtime Languages

## **Compile-Time Language**

* Compile-time languages, such as C and C++, are languages where the source code is translated into machine code by a compiler before execution. During compilation, the compiler analyzes the source code, checks for syntax errors, and translates it into an executable format that the computer's processor can understand. This process results in the creation of an executable file containing machine instructions that can be directly executed by the processor.
* One of the key advantages of compile-time languages is their efficiency. Since the code is translated into machine code beforehand, there is no need for interpretation or translation during runtime, leading to faster execution speeds. Additionally, compile-time languages typically offer fine-grained control over memory management and hardware resources, allowing developers to optimize their code for performance.
* However, compile-time languages also have some limitations. Because the code is translated into machine code before execution, any changes to the source code require recompilation of the entire program. This can be time-consuming, especially for large projects. Additionally, compile-time languages are often less portable than interpreted languages, as the compiled executable may only run on specific hardware architectures or operating systems.

## **Runtime Language**

* Runtime languages, such as Java and Python, are languages where the source code is translated into machine code and executed on-the-fly by an interpreter or a virtual machine during runtime. Instead of being compiled into native machine code, the source code is translated into an intermediate representation, such as bytecode, which is then executed by the interpreter or virtual machine.
* One of the main advantages of runtime languages is their portability. Since the source code is not compiled into native machine code, runtime languages can run on any platform that has an interpreter or virtual machine compatible with the bytecode format. This makes runtime languages well-suited for developing cross-platform applications.
* Additionally, runtime languages offer greater flexibility and ease of development. Developers can make changes to the source code and immediately see the results without the need for recompilation. This rapid iteration cycle can speed up the development process and make debugging easier.
* However, runtime languages typically have slower execution speeds compared to compile-time languages, as the code must be interpreted or translated by the virtual machine during runtime. This overhead can result in decreased performance, especially for computationally intensive applications. Additionally, runtime languages may have less control over hardware resources and memory management compared to compile-time languages.

# Compiler vs Interpreter

## **Compiler**

A compiler is a software tool that translates source code written in a high-level programming language into machine code or bytecode. The process involves several stages:

1. **Lexical Analysis:** The compiler scans the source code to identify and categorize lexemes or tokens, such as keywords, identifiers, and symbols.
2. **Syntax Analysis:** Also known as parsing, this stage analyzes the structure of the source code based on the grammar rules of the programming language. It verifies whether the code conforms to the language's syntax.
3. **Semantic Analysis:** The compiler performs checks to ensure that the code follows the semantics or meaning of the programming language. This includes type checking and scope resolution.
4. **Intermediate Code Generation:** Some compilers generate an intermediate representation of the source code, such as assembly language or bytecode, before producing the final machine code. This intermediate code can be optimized for better performance.
5. **Code Optimization:** In this phase, the compiler applies various optimization techniques to improve the efficiency and speed of the generated code. This can include dead code elimination, loop optimization, and register allocation.
6. **Code Generation:** Finally, the compiler translates the optimized intermediate code into machine code or bytecode suitable for execution on the target platform.

Once the compilation process is complete, the resulting executable file can be run independently of the compiler.

## **Interpreter**

An interpreter is another type of language translator that executes source code directly, without the need for prior compilation into machine code. Instead of producing an executable file, an interpreter reads the source code line by line and executes it on-the-fly.

The process of interpretation involves the following steps:

1. **Lexing and Parsing:** Similar to compilation, the interpreter first performs lexical analysis to tokenize the source code and then parses it to determine its syntactic structure.
2. **Execution:** Unlike compilers, interpreters execute the code immediately after parsing each statement or expression. They evaluate expressions, execute control flow structures (such as loops and conditionals), and manipulate data in memory as directed by the source code.
3. **Dynamic Typing:** Interpreters often support dynamic typing, where variable types are determined at runtime rather than compile-time. This allows for more flexibility but may also introduce runtime errors if type inconsistencies occur.

Interpreters are commonly used in scripting languages like **Python**, **JavaScript**, and **Ruby**, where rapid development and platform independence are prioritized over raw execution speed. While interpreted languages may sacrifice some performance compared to compiled languages, they offer advantages in terms of portability and ease of debugging.

Chapter 2

Basic of Programming (Java)

# **Editions of Java**

These are some of the editions of Java

## **Java SE (Standard Edition)**

Overview Java SE is the core Java platform, providing the foundation for developing and running Java applications on desktops and servers. It includes the Java Development Kit (JDK), which consists of the Java Runtime Environment (JRE), compiler, and other tools needed for Java development.

**Key Features**

1. **Java Development Kit (JDK):** The JDK is a software package that includes tools such as the Java compiler (javac), the Java Virtual Machine (JVM), and various utilities for developing and running Java applications. It also includes libraries and documentation.
2. **Java Runtime Environment (JRE):** The JRE is a subset of the JDK and contains only the runtime components necessary to run Java applications. It includes the JVM and core class libraries.
3. **Core Libraries:** Java Standard Edition provides a rich set of core libraries that offer functionalities for data structures, input/output operations, networking, graphical user interfaces (GUIs),java.lang, java.net and more. These libraries help developers build robust and efficient applications.
4. **Java APIs:** Java SE includes a vast collection of APIs that developers can use to access various functionalities. These APIs cover areas like networking, file handling, concurrency, security, and more.
5. **Language Features:** Java SE introduces the Java programming language itself, with features such as object-oriented programming, **platform independence**, and strong type checking.
6. **Compatibility:** Java SE emphasizes backward compatibility, allowing older Java applications to run on newer versions of the platform without major modifications.
7. Tools javac (compiler), java (interpreter), javadoc (documentation generator), etc.

Usage Java SE is used for a wide range of applications, including desktop GUI applications, command line tools, web servers, middleware, and more. It's the foundation for building Java based software across various industries and domains.

## **Java ME (Micro Edition)**

Overview Java ME is a stripped down version of Java SE designed for resource constrained devices such as mobile phones, PDAs, settop boxes, and embedded systems. It provides a runtime environment and APIs tailored for small devices with limited memory, processing power, and display capabilities.

**Key Features**

1. Configurations and profiles Java ME consists of different configurations (e.g., Connected Limited Device Configuration **CLDC**) and profiles (e.g., Mobile Information Device Profile **MIDP**) (e.g., javax.microedition.lcdui, optimized for specific device categories.

* **GPIO** (General Purpose Input Output) applications.

Usage Java ME was widely used for developing mobile applications and games before the rise of smartphones. It's still used in some IoT (Internet of Things) and embedded systems applications.

## **Java EE (Enterprise Edition)**

Java EE stands for Enterprise Edition. It was developed with the aim of extending Java SE by adding a collection of standards/specifications that define frequently-used features by commercial applications.

The latest version of this Java edition includes over 40 specifications that help programmers build applications that use web services to transform object-relational data into entity-relationship models, exchange data, manage transactional interactions, and more.

Java EE is a platform for building enterprisescale applications, providing a set of standardized APIs and runtime environments for developing distributed, scalable, and secure software systems.

**Key Features**

1. Enterprise APIs Java EE includes APIs for web services (**JAXWS**, **JAXRS**), persistence (**JPA**), messaging (**JMS**), dependency injection (**CDI**), security (**JAAS**), transactions (**JTA**), etc.
2. **Java Persistence API (JPA)** It allows Java developers to access an object/relational mapping mechanism so they can manage relational data in Java applications.
3. **Java Server Pages (JSP)** is a server-side programming language used to create device-independent and dynamic ways to build web applications.
4. **Enterprise Java Beans (EJB)** It is among the Java APIs that are used for the development of corporate applications. EJB is a server-side software component that describes the business logic of an application.
5. **Java Server Faces (JSF)** API provides elements, such as commandButtons, inputText, and more, and assists in managing their states. It also offers server-side validation, data translation, etc.
6. Application servers Java EE applications run on compliant application servers *(e.g., Apache Tomcat, JBoss, IBM WebSphere)*, which provide runtime support for Java EE features.

Usage Java EE is used for developing a wide range of enterprise applications, including web applications, ecommerce systems, financial services applications, CRM (Customer Relationship Management) systems, and more. It's especially suited for largescale, mission critical applications requiring reliability, scalability, and security.

## **Other Editions**

1. **JavaFX** is a platform for creating rich internet applications (**RIAs**) and multimedia applications using Java. It provides a set of APIs for building GUIs, animations, media playback, and more. It provides a modern, hardware-accelerated graphics and media engine for developing rich desktop applications. We can use JavaFX Script, a simple yet powerful scripting language, to enable the development of rich online applications, desktop applications, and GUI applications. Java created it to replace Swing as the default GUI library.

* **GEONS** Ground System Software (**GGSS**) Nasa is used in space technology.
* **NEOS** (New Eurovision Operations System) is used in television media.
* **FORUM** Carl Zeiss Meditec AG is used in the field of medicine.
* Quote Monitor application is used in the finance sector.

1. **Java Card** is a specialized version of Java for smart cards and other small memory footprint devices. It provides a runtime environment and APIs for developing secure applications for smart cards used in banking, telecommunications, identity, and access control systems.

Each edition of Java serves specific purposes and target platforms, catering to diverse application development needs across different domains and industries.

# **JVM (Java Virtual Machine)**

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.

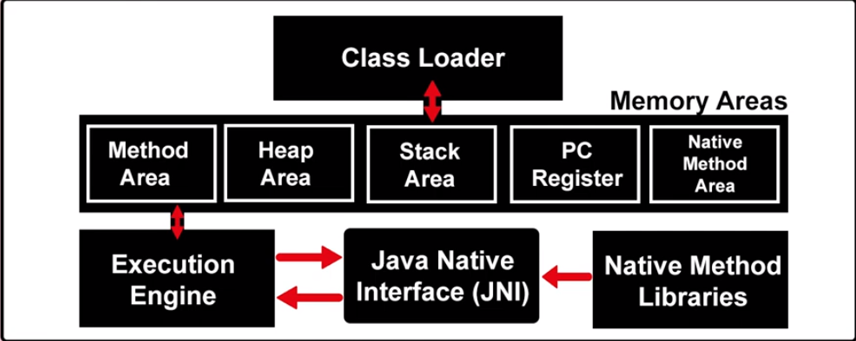
**When JVM starts it performs some specific tasks**

* 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**



**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.

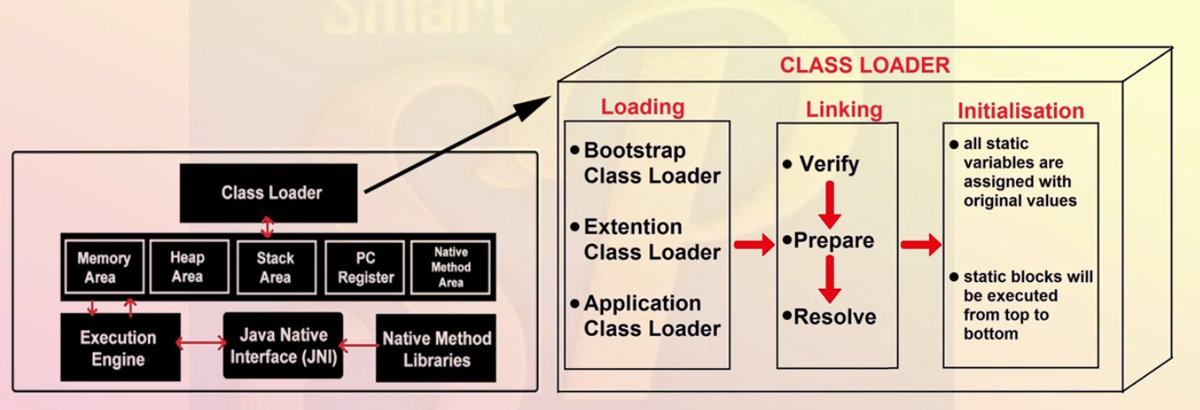
1. Types of virtual machine:

* 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*

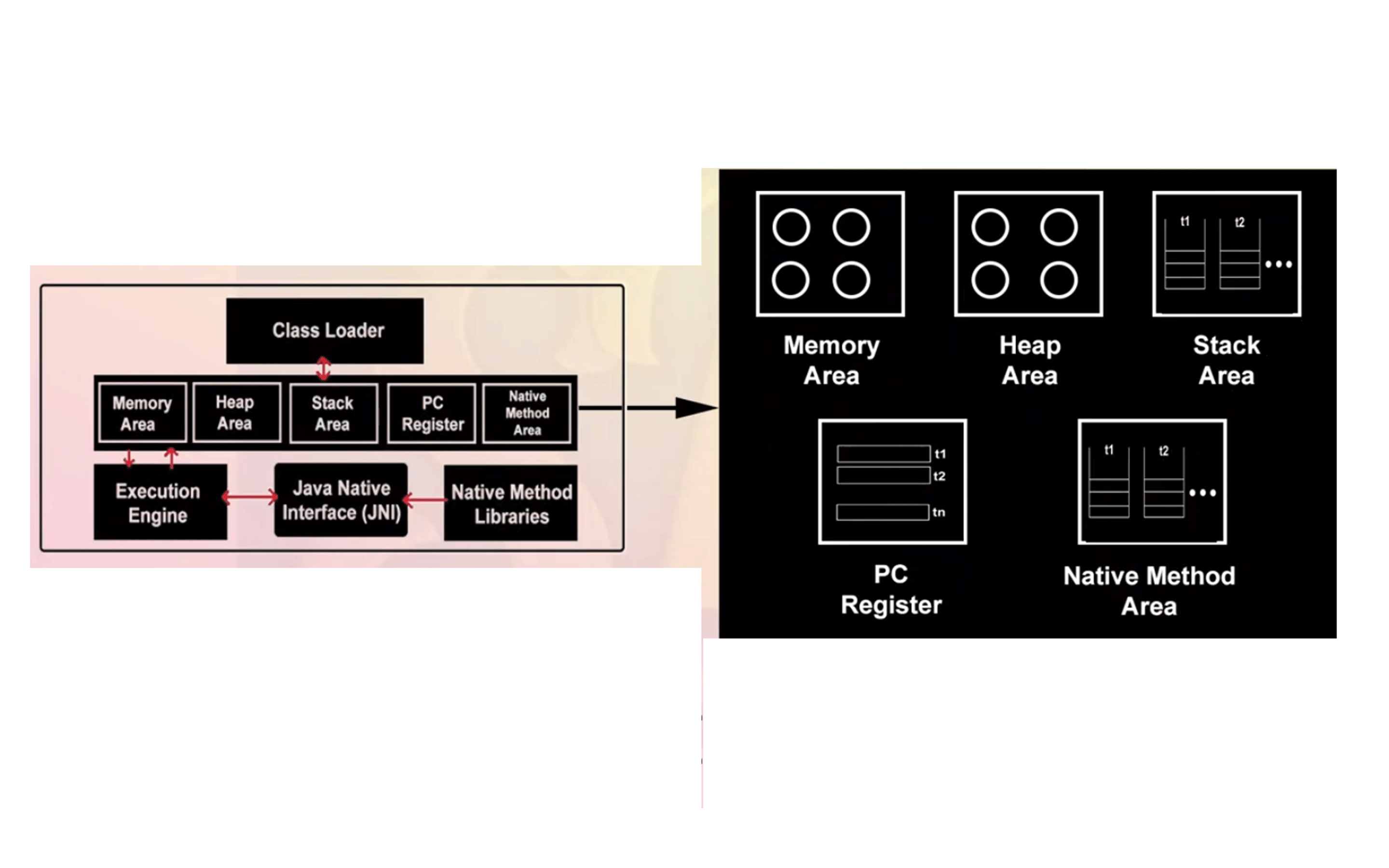
* **Verification** In this process Byte Code Verifier Checks Whether the -class file is generated by valid compiler or not and whether \*Class file is properly formatted or not. If verification fails, then JVM will provide "**java.Iang.VerifyError**" exception. Because of this process, java is secured.
* **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

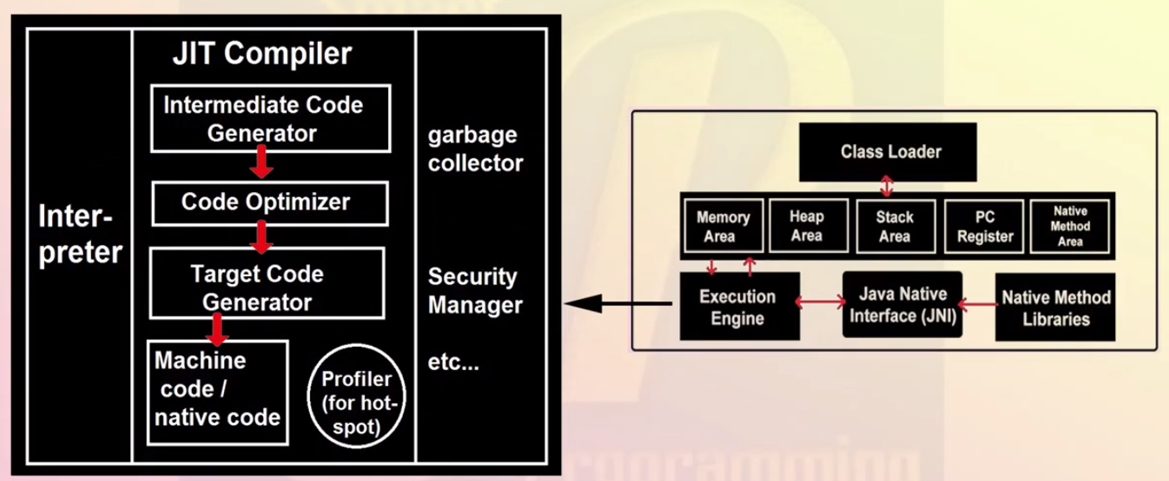
1. All static variables are assigned with original values.
2. static blocks will be executed from top to bottom

### **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.

# **JDK (Java Development Kit)**

A diagram of a program

Description automatically generated with medium confidence

|  |
| --- |
| Java Developer Kit contains tools needed to develop the Java programs, and JRE to run the programs.  JDK is targeted for java development. i.e. You can create a Java file (with the help of Java packages), compile a Java file and run a java file.  **JDK = JRE + Development Tools** |

**Task of JDK**

1. **Development Environment**: JDK is a software development kit that provides tools for developing Java applications.
2. **Includes JRE**: JDK includes everything that's in JRE, plus additional development tools like compilers (javac), debuggers (jdb), and documentation.
3. **Versions**: JDK versions are released by Oracle (and other vendors) and include updates, bug fixes, and enhancements to the Java platform.
4. **Platform Independence**: JDK allows developers to write Java code on one platform and run it on any other platform where JRE is installed, ensuring platform independence.
5. **Versions and Compatibility**: JDK versions are backward compatible, meaning code written in older versions of Java can usually run on newer JDK versions without modification.

## **JDK Tools**

**javac** Java compiler CompilCompiler

Compiles Java source code (.java files) into bytecode (.class files) that can be executed by the JVM.

**01**

**java** Java launcher CompilCompiler

Launches Java applications by interpreting bytecode and executing it on the Java Virtual Machine (JVM).

**02\2**

**javadoc** Java Document generator CompilCompiler

Generates API documentation in HTML format from Java source code comments.

**03\2**

**javap** Java Class File Disassembler

Disassembles bytecode (.class files) into human-readable Java assembly code.

**05\2**

**jdb** Java Dibugger

Command-line debugger for debugging Java applications, allowing developers to step through code, set breakpoints, and inspect variables.

**04\2**

**jarsigner**JAR File Signer   
and Verifier

Signs and verifies Java Archive (JAR) files using digital certificates for security purposes.

**07\2**

**jvisualvm**Java VisualVM

Visual tool for monitoring, troubleshooting, and profiling Java applications, providing insights into memory usage, CPU usage, and thread activity.

**09**

**jlink** Java Linker

Graphical tool for monitoring and managing Java applications, including memory usage, thread activity, and performance.

**10**

**jconsole** Java Monitoring and  
Management Console

Graphical tool for monitoring and managing Java applications, including memory usage, thread activity, and performance.

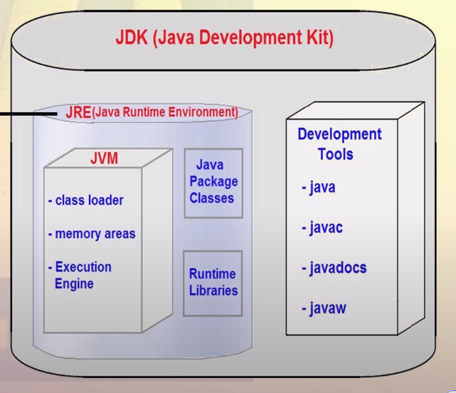
**08**

**jdeps**  Java Dependency Analysis Tool

Analyses Java class files or JAR files to determine dependencies between classes and packages.

**06\2**

# **JRE (Java Runtime Environment)**

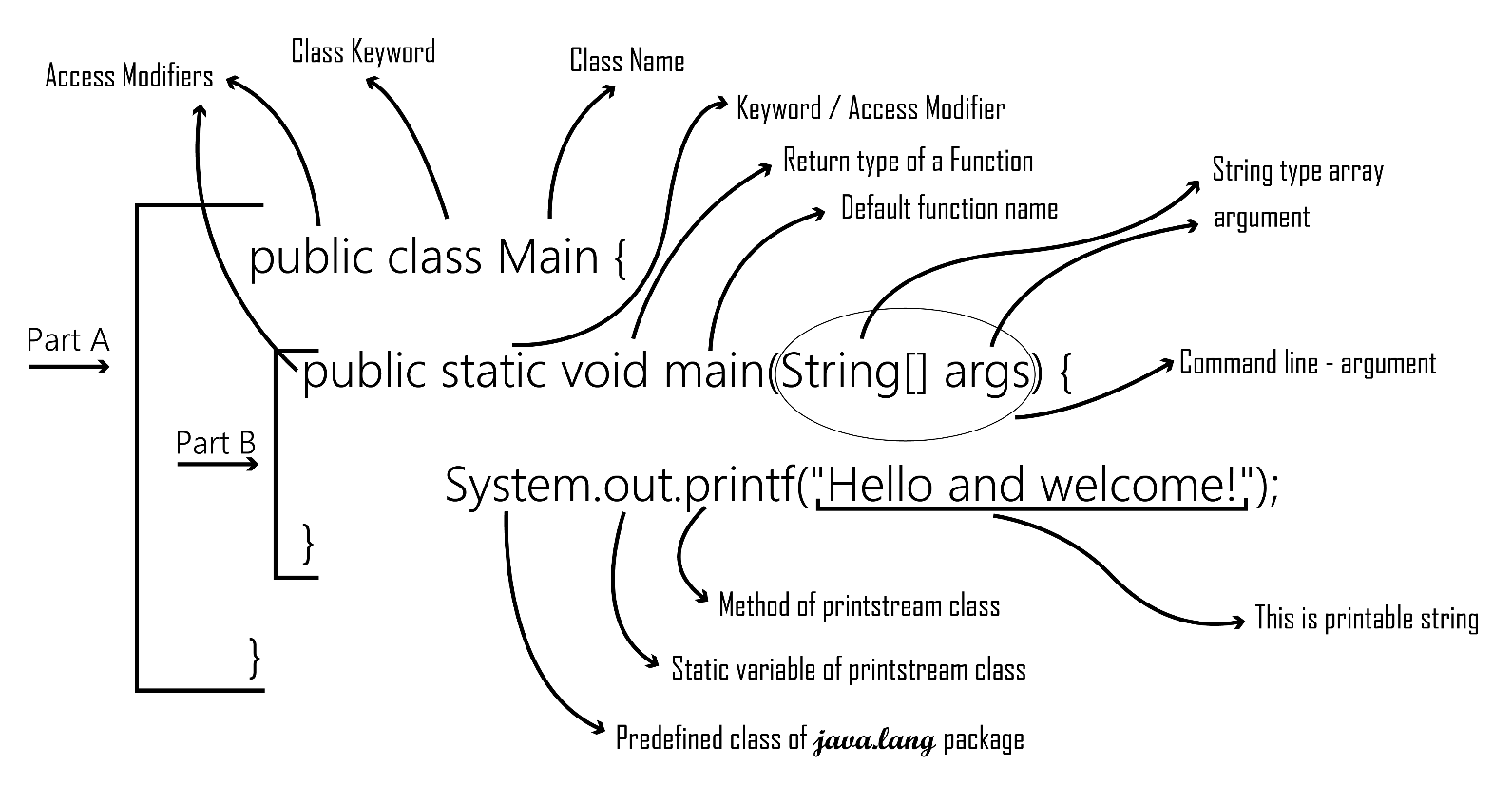


|  |
| --- |
| Java Runtime Environment contains JVM, class libraries, and other supporting files. It does not contain any development tools such as compiler, debugger, etc. JVM runs the program, and it uses the class libraries, and other supporting files provided in JRE. If you want to run any java program, you need to have JRE installed in the system.  **JRE = JVM + Java Package Classes + Runtime Libraries** |

**Task of JDK**

1. **Execution Environment**: JRE is a runtime environment that enables the execution of Java applications on a computer system.
2. **Components**: It includes the Java Virtual Machine (JVM), core libraries, and other supporting files necessary for running Java applications.
3. **No Development Tools**: JRE does not include development tools like compilers and debuggers; it is meant purely for running Java applications.
4. **Distribution**: JRE is often distributed alongside Java applications to ensure compatibility and proper execution.

# **Main Method**



Before explaining main method, we separate main method in two parts for better understanding **class part** or **method part** to understand the concept easily.

1. **Class part**
2. **Class**
3. class is a keyword.
4. Java is an object-oriented programming, and you cannot run your code without a class in Java. So, you need to put your code inside a class.
5. In one java program, there can be only single public class.
6. **Syntax of class :**

access-modifier class ClassName {

//body

}

1. **Syntax of Main Method**

// Syntax

*access-modifier static return\_type function\_name (String [] args){*

*// Method body*

*}*

// Actual Method

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

*System.out.println(“Hello Java Learner”);*

*}*

1. If a class has no modifier (the default, also known as package-private), it is visible
2. only within its own package (you will learn about them in a later episodes)
3. **Main** is an Identifier (user-defined class name).
   1. This is the name of the class. In java, the class name must match the filename and start with an uppercase letter.
4. Class-name should follow the rules of an identifier.
5. **Public**
6. public is access modifier keyword.
7. It indicates that the class “Main” is accessible form any other class. Other access modifiers include private, protected and package private (default).
8. JVM can be installed anywhere i.e. C or D or any other drive and JVM is responsible to call main method which may be in any other drive, thus, to call main method by JVM from anywhere, "main" method should be declared public.
9. **Main method**
   1. **Public**

Another access modifier, indicating that the method ‘**main’** can be accessed from any other class.

* 1. **Static**

1. static is keyword indicates that the method belongs to the class itself, rather than to any specific instance of the class.
2. At the time of execution there is no object, thus by declaring main method as static, it becomes class level method (not object level method), so it can be called without object (JVM class main method). Thus main method nowhere related to object.
3. **Void**
4. void is return type keyword.
5. void means nothing or empty.
6. main method has void return type as it does not return anything to JVM.
7. If it returns then what will JVM do with the value, it will be of no use.
8. **Main()**
9. main is predefined method, which is configured inside JVM.
10. This is the name of the method.
11. In java the main method serves as the entry point of for the program.
12. JVM always Start its execution from main method.
13. **String[] args**
14. Command-line arguments.
15. This is the parameter list of the method. Here ‘**args’** is an array of strings (‘String []'), which allows command-line arguments to be passed to the program when it is executed.
16. **System**
17. Predefined class present in “java.lang” package.
18. This is a class in java, representing the standard input, output and error streams.
19. It provides access to system resources.
20. **Out**
21. Out is a “static variable” of “PrintStream” class present in “System” class.
22. This is a public static member of the system class.
23. It is an instance of the ‘PrintStream’ class
24. Three static variables of ‘PrintStream’ class are:

* **out** – for output { System.out.println(“Output 1”); }
* **in** – for input { Scanner sc = new Scanner(System.in); }
* **err** - for error { System.err.println(“Output 3”); }

1. **Println()**
2. Println() is method of PrintStream class. Used to print a string followed by a newline character.
3. Three different methods of PrintStream class to print n console are

* **print**()
* **println**()
* **printf**()

# **Types of Errors**

1. **Lexical Error**

* **Definition**: Lexical errors, also known as lexical analysis errors or scanning errors, occur when the compiler encounters illegal characters or sequences of characters in the source code that do not conform to the lexical rules of the programming language.
* Examples: Misspelled keywords, invalid characters in identifiers, missing or misplaced punctuation, etc.

1. **Syntax Error**

* **Definition**: Syntax errors occur when the compiler encounters code that violates the grammatical rules (syntax) of the programming language. These errors prevent the compiler from parsing the code correctly.
* Examples: Missing semicolons, mismatched parentheses, incorrect syntax for control flow statements (if, else, while, for), etc.

1. **Semantic Error**

* **Definition**: Semantic errors, also known as semantic analysis errors, occur when the code compiles and executes without syntax errors but does not produce the expected result due to incorrect logic or misunderstanding of the programming language semantics.
* Examples: Using variables without initializing them, performing operations on incompatible data types, incorrect algorithm implementation, etc.

1. **Logical Error**

* **Definition**: Logical errors occur when the code executes without syntax or semantic errors but produces incorrect results due to flawed logic or incorrect assumptions in the code.
* Examples: Off-by-one errors in loops, incorrect conditional statements, wrong algorithm design, etc.

1. **Type Error**

* **Definition**: Type errors occur when operations are performed on data types that are incompatible or when the expected type of data does not match the actual type.
* Examples: Trying to add a string and an integer without proper conversion, passing the wrong type of argument to a method, etc.

1. **Runtime Error**

* **Definition**: Runtime errors, also known as exceptions or execution-time errors, occur while the program is running. These errors can result from various issues such as division by zero, array index out of bounds, null pointer dereference, etc.

1. **Overflow/Underflow Error**

* **Definition**: Overflow and underflow errors occur when arithmetic operations result in a value that is too large (overflow) or too small (underflow) to be represented by the data type.
* Examples: Adding two large integers that exceed the maximum value for the data type, dividing by a very small number resulting in underflow, etc.

1. **Concurrency Error**

* **Definition**: Concurrency errors occur in multithreaded or parallel programs when multiple threads or processes access shared resources concurrently, leading to unexpected behavior or data corruption.
* Examples: Race conditions, deadlock, livelock, thread synchronization issues, etc.

# **Variables**

In Java, Variables are the data containers that save the data values during Java program execution. Every Variable in Java is assigned a data type that designates the type and quantity of value it can hold. A variable is a memory location name for the data.

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

Declaring a variable

Syntax

**datatype** **variableName**;

Example

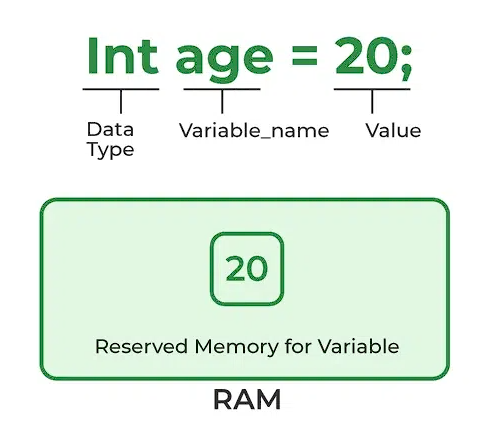
**int** number;

*// Here number is a variable of int type which can only be stored integer number only. A specific type of variable can only be stored specific type of data like an string can store only string, an double type of variable can store double type value, an Boolean type variable can store only true or false, and character type variable can store only character type value.*

**Variable Initialization**

In this way, a name can only be given to a memory location. It can be assigned values in two ways:

* Variable Initialization
* Assigning value by taking input



Initialization of a Variable

* **datatype**: Type of data that can be stored in this variable.
* **variable\_** Name given to the variable.
* **value**: It is the initial value stored in the variable.

There are three main types of variables in Java:

1. **Local Variables**
   * **Scope**: Local variables are visible only within the block, method, or constructor in which they are declared. They exist only for the duration of that block, method, or constructor's execution.
     + These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function
   * **Accessibility**: They cannot be accessed from outside the method, constructor, or block where they are declared. They are essentially private to the code block in which they are defined.
     + Initialization of the local variable is mandatory before using it in the defined scope.
     + Access modifiers are not allowed for local variable.
   * **Usage**: Local variables are commonly used for temporary storage of data within methods or blocks. They are not suitable for storing data that needs to be accessed outside of the block where they are defined.

public void exampleMethod() {

int localVar = 10; // localVar is a local variable

}

**Example 1**

let's consider this example to demonstrate the usage of local variables:

public class Car {

public void printModel(String model) {

System.out.println("Model of the car: " + model); // local variable

}

}

In this `Car` class, we have a method called `printModel()` which takes a `model` parameter as a local variable.

Now, let's use this method in the `Main` class:

public class Main {

public static void main(String[] args) {

Car car = new Car();

String model = "Toyota";

car.printModel(model);

}

}

**In this example:**

* + - We create an instance of the `Car` class.
    - We define a local variable `model` in the `main()` method and initialize it with the value `"Toyota"`.
    - We call the `printModel()` method of the `Car` class and pass the `model` variable as an argument.
    - Inside the `printModel()` method, `model` is a local variable that holds the value passed from the `main()` method.

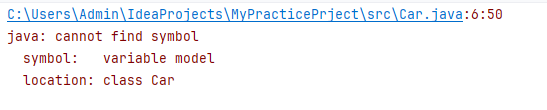
Local variables have a limited scope and lifetime. In this example, the `model` variable exists only within the `main()` method. Once the `main()` method execution completes, the `model` variable goes out of scope and is eligible for garbage collection. Each time the `printModel()` method is called, a new instance of the `model` variable is created with the value passed to the method, but it is independent of any other invocations of the method.

**Example 2**

But what if we print the local variable out of the scope. Let’s understand this using another program.

public class Car {  
 public void printModel1(String model){  
 System.*out*.println("Model of car is -> "+model);  
 }  
 public void printModel2(){  
 System.*out*.println("Model of car is -> "+model);  
 }  
}  
  
class Main{  
 public static void main(String ...args){  
 Car carObj = new Car();  
 *// method 1* carObj.printModel1("Toyota Supra");  
  
 *// method 2* carObj.printModel2();  
 }  
}

**Explanation**

* + **Classes and Methods**: This program consists of two classes: Car and Main. The Car class contains two methods: printModel1 and printModel2. The first method (printModel1) contains a local variable called model, while the second method (printModel2) does not contain any variables.
  + **Scope of Local Variables**: When we create an object of the Car class and set a value using printModel1, which has a local variable model, the provided value is printed using an output statement. However, if we attempt to print the same value in printModel2, the Java compiler will throw an error. This is because the scope of the model variable is limited to printModel1 and cannot be accessed by printModel2 or outside its boundaries.
  + **Understanding Local Variable Scope**: This program illustrates the concept of local variable scope and lifetime. Local variables are accessible only within the method or block in which they are declared. Once the method execution completes, the local variables cease to exist, and their values cannot be accessed from outside their scope. This demonstrates the importance of understanding variable scope in Java programming.

1. **Instance Variables (Non-Static Variables)**
   * **Scope**: Instance variables (non-static) are accessible throughout the entire class in which they are declared. They persist for the lifetime of the object to which they belong.
     + These variables are declared within a class but outside any method, constructor, or block. Each instance of the class (i.e., each object) has its own copy of instance variables.
     + These variables are created when an object of the class is created and destroyed when the object is destroyed.
   * **Accessibility**: They can be accessed by any method or constructor within the class, as well as by methods of other classes if they are public or if accessor methods (getters and setters) are provided.
     + Unlike local variables, we may use access specifiers for instance variables like *public*, *protected*, *private*. If we do not specify any access specifier, then the default access specifier will be used.
     + Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable. For String it is null, for float it is 0.0f, for int it is 0, for Wrapper classes like Integer it is null, etc.
     + Instance variables can be accessed only by creating objects.
     + We initialize instance variables using constructors while creating an object. We can also use instance blocks to initialize the instance variables.
   * **Usage**: Instance variables are used to store data that is unique to each object of the class. They hold the state of the object and are accessed and modified by methods of the class.

public class MyClass {

int instanceVar = 20; // instanceVar is an instance variable

}

**Example 1**

let's consider a simple example to demonstrate instance variables and how their copies are different for each object.

Suppose we have a `**Car**` class with an instance variable `*model*`:

public class Car {

String model; // instance variable

// setter method used for setting values to instance variable

public void setModel(String model) {

this.model = model;

}

// getter method used of getting the value of instance variable

public String getModel() {

return model;

}

}

Now, let's create two `Car` objects and set different models for each:

public class Main {

public static void main(String[] args) {

Car car1 = new Car();

Car car2 = new Car();

car1.setModel("Toyota");

car2.setModel("Honda");

System.out.println("Model of car1: " + car1.getModel());

// Output: Model of car1: Toyota

System.out.println("Model of car2: " + car2.getModel());

// Output: Model of car2: Honda

}

}

In this example:

1. We have a class `**Car**` with an instance variable `*model*`.
2. We create two `**Car**` objects, `*car1*` and `*car2*`.
3. We set different models for each car using the `**setModel()**` method.
4. When we print the models of `*car1*` and `*car2*`, we see that they have different values.

This demonstrates that each instance of the `**Car**` class (`*car1*` and `*car2*`) has its own copy of the `model` variable. Modifying the `model` variable of one `**Car**` object does not affect the `model` variable of another `**Car**` object. Each object maintains its own state independent of other objects, which is a fundamental concept of object-oriented programming and the use of instance variables.

1. **Static Variables (Class Variables)**  
   These variables are declared with the **static** keyword within a class but outside any *method*, *constructor*, or *block*. They are associated with the class rather than with any object.
   * **Scope**: Static variables have class scope, meaning they are accessible throughout the entire class in which they are declared. They exist independently of any object of the class.
     + There is only one copy of a static variable per class, regardless of how many objects are created from it.
     + Static variables are also known as class variables.
   * **Accessibility**: They can be accessed by any method or constructor within the class, including static methods. They can also be accessed directly using the class name from outside the class, provided they are declared with an appropriate access modifier (usually public or protected).
     + Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
     + Static variables are created at the start of program execution and destroyed automatically when execution ends.
     + Initialization of a static variable is not mandatory. Its default value is dependent on the data type of variable.
       - For String it is null, for float it is 0.0f, for int it is 0, for Wrapper classes like Integer it is null, etc.
     + If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won’t halt the program. The compiler will replace the object name with the class name automatically.
     + If we access a static variable without the class name, the compiler will automatically append the class name. But for accessing the static variable of a different class, we must mention the class name as 2 different classes might have a static variable with the same name.
     + Static variables cannot be declared locally inside an instance method.
     + Static blocks can be used to initialize static variables.
   * **Usage**: Static variables are used to store data that is shared among all instances of the class. They are commonly used for constants, configuration values, or to maintain global state across instances of the class.

public class MyClass {

static int staticVar = 30; // staticVar is a static variable

}

Additionally, Java has a special type of variable called **final variables**. Once a final variable is assigned a value, it cannot be changed. Final variables can be class variables, instance variables, or local variables.

public class MyClass {

final int finalVar = 40; // finalVar is a final instance variable

public void exampleMethod() {

final int localFinalVar = 50; // localFinalVar is a final local variable

}

static final int staticFinalVar = 60; // staticFinalVar is a final static variable

}

**Instance Variable VS Static Variable**

Now let us discuss the differences between the Instance variables and the Static variables:

* Each object will have its own copy of an instance variable, whereas we can only have one copy of a static variable per class, irrespective of how many objects we create. Thus, static variables are good for memory management.
* Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of the instance variable. In the case of a static variable, changes will be reflected in other objects as static variables are common to all objects of a class.
* We can access instance variables through object references, and static variables can be accessed directly using the class name.
* Instance variables are created when an object is created with the use of the keyword ‘new’ and destroyed when the object is destroyed. Static variables are created when the program starts and destroyed when the program stops.

# **Datatype**

## Unicode system

# **Tokens**

In Java, a token is the smallest unit of a program that is meaningful to the compiler. Tokens are the building blocks of a Java program and include keywords, identifiers, literals, operators, separators, and comments. Let's explore each type of token in Java along with examples:

1. **Keywords**
   1. Keywords are reserved words that have predefined meanings in Java and cannot be used as identifiers.
   2. In Java, there are a total of 68 reserved keywords defined in java language
   3. The java language limits the use of its keywords and divides them into two categories

* Reserved keywords (58)
* Contextual keywords (17)
  1. Ten keywords related to modules
  2. These keywords have predefined meanings and cannot be used as identifiers such as
* Variable names
* Method names
* Class names

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| abstract | catch | do | final | implements | module |
| private | short | this | try | assert | char |
| double | finally | import | native | protected | static |
| throw | uses | boolean | class | else | float |
| instanceof | new | provides | strictfp | throws | void |
| break | const | enum | for | int | open |
| public | super | to | volatile | byte | continue |
| exports | goto | interface | opens | requires | switch |
| transient | while | case | default | extends | if |
| long | package | return | synchronized | transitive | With |
| \_ |  |  |  |  |  |

Sir ji total 68 keywords he lakin total counting 69 aa rahi he app dheek lena ek brr

Separates all of them accordingly to its works

1. **Primitive Type Keywords (8)**
2. byte
3. short
4. int
5. long
6. float
7. double
8. char
9. boolean
10. **Flow Control Keywords (11)**
11. if
12. else
13. switch (supports local variable type inference)
14. case
15. default
16. while
17. do
18. for
19. break
20. continue
21. return
22. **Access Modifier Keywords (3)**
23. public
24. private
25. protected
26. **Class and Object Keywords (10)**
27. **Class Related Keywords (4)**
    1. class
    2. interface
    3. extends
    4. implements

**Package Related Keywords (2)**

1. package
2. import
3. **Object Related Keywords(4)**
   1. new
   2. this
   3. super
   4. instanceof
4. **Method-related Keywords (7)**
5. void
6. static
7. final
8. synchronized
9. abstract
10. native
11. strictfp (it has been rendered obsolete because it has no effect)
12. **Exception Handling Keywords (5)**
13. try
14. catch
15. finally
16. throw
17. throws
18. **Other Keywords (6)**
19. true
20. false
21. null
22. assert
23. const (Reserved for future use, not used in modern Java)
24. goto
25. **Contextual Keywords (14)**
26. enum
27. exports
28. open
29. opens
30. module
31. provides
32. requires
33. to
34. with
35. uses
36. record
37. sealed
38. non-sealed
39. permits
40. **Reserved in Java 9 and Later (5)**
41. Var (supports local variable type inference)
42. transient
43. volatile
44. yield
45. when

These keywords play crucial roles in defining the structure, behavior, and features of Java programs Understanding their usage and significance is essential for writing correct and efficient Java code

1. **Identifiers**
   * Identifiers are names given to classes, variables, methods, packages, etc., in a Java program.
   * **Rules for identifiers**:
     + The only allowed characters in java identifiers are a to z, A to Z, O to 9, $ and \_ (Underscore). If we are using any other character then we will get compilation error.
     + Identifiers can't start With digit i.e., 123name is not valid.
     + Java identifiers are case sensitive of course java language itself considered as case sensitive programming language. i.e., name, Name, NAME both are different variable names.
     + There is no length limit for java identifiers but it is not recommended to take too length identifiers.
     + We can't use reserved words as identifiers otherwise we will get compilation error. i.e., int if — 20; here we will get compilation error.
     + All predefined java class names and interface names we can use as identifiers. i.e., int String = 20; here we won't get any compilation error. But it is not a good programming practice.
2. **Literals**
   * Literals represent constant values in Java code.
   * Types of literals:
     + **Integer literals:** e.g., 10, 0xFF, 123\_456.
     + **Floating-point literals:** e.g., 3.14, 2.5e-3.
     + **Boolean literals:** true or false.
     + **Character literals:** e.g., 'a', '9', '\n'.
     + **String literals:** e.g., "Hello, world!", "Java".
   * Note: Literals are direct representations of values in the code.
3. **Operators**
   * Operators perform operations on operands.
   * Types of operators: Arithmetic, relational, logical, assignment, bitwise, etc.
   * Examples: +, -, \*, / (arithmetic), ==, !=, <, > (relational), &&, ||, ! (logical), =, +=, -=, \*= (assignment), &, |, ^, ~ (bitwise), etc.
4. **Separators**
   * Separators are characters used to separate tokens in Java code.

|  |  |  |
| --- | --- | --- |
| **Symbol name** | **Symbol Sign** | **Working** |
| **Braces** | **{}** | Used to define the scope of classes, methods, loops, and conditional statements in Java. |
| **parentheses** | **()** | Used to enclose parameters in method declarations, control flow expressions, and to control the order of operations in arithmetic expressions. |
| **brackets** | **[]** | Used to declare arrays in Java, access elements of arrays, and define array index expressions |
| **commas** | **,** | Used to separate items in lists, such as method parameters, array elements, and variable declarations. |
| **semicolons** | **;** | Used to terminate statements in Java, indicating the end of a line of code. |
| **periods** | **.** | Used to access members (fields or methods) of classes and objects, as well as to specify the package hierarchy in fully qualified class names. |
| **colons** | **:** | Used in various contexts, such as the ternary conditional operator (condition ? expr1 : expr2), labeled statements, and in the syntax of enhanced for loops (for (type element : collection)). |
| **Ellipsis** | **…** | Used in method declarations to denote varargs (variable-length argument lists), allowing methods to accept a variable number of arguments of the same type. |
| **Ampersand** | **@** | Used as the bitwise AND operator, logical AND operator, and in Java 7 and later as the "address-of" operator for lambda expressions |

1. **Comments**
   * Comments are used to add explanations or notes within the code for developers and are ignored by the compiler.
   * Types of comments: Single-line comments //, multi-line comments **/\* \*/**, and documentation comments **/\*\* \*/**.
   * Examples:
     1. **Single-line comment**
        1. Single-line comments start with // and continue until the end of the line.
        2. They are used to add short explanations or notes on a single line.
        3. Single-line comments are commonly used for code annotations or to temporarily disable code.

// This is a single-line comment

int age = 30; // Variable 'age' stores the age of the user

* + 1. **Multi-line comment**
       1. Multi-line comments start with /\* and end with \*/. They can span multiple lines.
       2. They are used to add longer explanations, comments, or block comments.
       3. Multi-line comments are useful for commenting out large sections of code or for adding detailed comments.

/\*

\* This is a multi-line comment

\* It can span multiple lines

\* It is useful for adding detailed explanations or comments

\*/

* + 1. **Documentation comment:**
       1. Documentation comments start with /\*\* and end with \*/. They are also used to span multiple lines.
       2. They are specifically formatted comments used for generating API documentation.
       3. Documentation comments support special tags like **@param**, **@return**, **@throws**, etc., to provide additional information about methods, classes, and fields.
       4. They are commonly used for documenting classes, methods, and interfaces to generate well-formatted documentation using tools like Javadoc.

/\*\*

\* This is a documentation comment for the MyClass class.

\* It provides information about the class, its purpose, and usage.

\*

\* @author John Doe

\* @version 1.0

\*/

public class MyClass {

/\*\*

\* This method adds two integers.

\*

\* @param a The first integer

\* @param b The second integer

\* @return The sum of the two integers

\*/

public int add(int a, int b) {

return a + b;

}

}