VCODEZ
INNOVATING IDEAS

LEARN JAVA PROGRAMMING

WRITE ONCE, INNOVATE EVERYWHERE!

Step into the world of robust programming with Java. Dive deep into the language that powers modern applications and drives global innovation .

CONTACT US ➤ 9600189201 info@vcodez.com

TABLE OF CONTENT

MYSQL

- Making HTTP Requests
- Observables and RxJS
- Handling HTTP Responses and Errors
- HTTP Interceptors

WEB APP &AMP;JDBC

- ➢ Heading
- ➢ Introduction
- ➢ Java EE Layers
- ➢ Servlet Request-Response Cycle
- ➢ JDBC Flow
- ➢ JSP in MVC Architecture
- ➢ Java EE Application Workflow
- ➢ JDBC (Java Database Connectivity)
- ➢ Components of JDBC
- ➢ Types of JDBC Drivers
- ➢ Establishing a Database Connection
- ➢ JDBC API Methods with Examples
- ➢ Best Practices for JDBC Programming

JAVA WEB APPLICATION

- ➢ Spring Framework
- ➢ Modern Web Application Characteristics
- ➢ History of Java Web Development
- ➢ Spring Boot
- ➢ Layers of a Web Application
- ➢ Design Patterns & Principles
- ➢ Bean in Spring Boot

- Inversion of Control (IoC)
- Dependency Injection (DI)
- Injection Types
- MVC with Spring Boot
- POJO Classes in Spring Boot
- Entity Representation, DTOs, Dependency Injection
- Spring Boot Annotations
- CRUD Application Setup
- JPA in Spring Boot
- Introduction to REST API
- Spring Boot REST Controller
- Key Features of REST API

INTRODUCTION TO JAVA

Java is a highly versatile, secure, and platform-independent programming language that has remained a cornerstone of software development since its release. It was developed by **Sun Microsystems**, led by James Gosling and his team, and officially released to the public on **May 23, 1995**. Java was created as part of a project initially called "Green" and was later named after Java coffee, reflecting its goal of being energizing and ubiquitous. Designed with the principle of "**Write Once, Run Anywhere**" (WORA), Java introduced a revolutionary approach to software development by enabling applications to run seamlessly across different platforms through the Java Virtual Machine (JVM).

Java is an **object-oriented language**, emphasizing modularity and reusability, making it ideal for developing complex, scalable systems. Its features, such as automatic memory management via garbage collection, robust exception handling, and built-in security, ensure high reliability and stability. Java's rich API library simplifies programming tasks ranging from networking and database connectivity to building graphical user interfaces (GUIs) and multithreaded applications.

Initially aimed at interactive TV systems, Java soon evolved to dominate areas such as web development, mobile applications (especially Android), enterprise solutions, gaming, and even Internet of Things (IoT) devices. Over the years, Java has maintained its relevance with regular updates and a thriving developer community, becoming one of the most widely used programming languages globally. Its release marked a significant milestone in computing history, and today it remains a vital tool for developers across industries.

JAVA VERSIONS

Java is a multi-faceted platform designed to cater to various domains in software development. Each edition of Java is tailored to address specific requirements, providing developers with the tools and libraries necessary for building diverse applications. Here’s a detailed breakdown of Java editions:

1. Java Standard Edition (Java SE)

Overview

Java Standard Edition (Java SE) is the foundation of the Java platform, offering the core tools and libraries needed for developing general-purpose applications. It includes the Java programming language, essential APIs, and the Java Virtual Machine (JVM), which ensures cross-platform execution. Java SE is the building block for all other editions and serves as the entry point for most developers.

Key Features

- **Core APIs**: Provides libraries for fundamental programming needs like collections, networking, I/O, multithreading, and exception handling.
- **Java Virtual Machine (JVM)**: The runtime engine that enables Java applications to run on any operating system.
- **Tools for Development**: Comes with the Java Development Kit (JDK), which includes:
    - A compiler for converting source code into bytecode.
    - Debugging tools for identifying and resolving issues.
    - The Java Runtime Environment (JRE) for running compiled programs.
- **Security**: Offers built-in mechanisms like cryptography and secure coding practices.

Applications

- Desktop software.
- Command-line utilities.
- Applications requiring portability across platforms.

Examples

- IDEs like NetBeans and Eclipse.
- Desktop productivity tools like file managers and media players.

---

2. Java Enterprise Edition (Java EE)

Overview

Java Enterprise Edition (Java EE), now rebranded as **Jakarta EE**, extends Java SE by providing a robust framework for developing distributed, scalable, and secure enterprise-level applications. Designed for large organizations, it supports the creation of web, server-side, and cloud-based applications that handle complex workflows and high traffic.

Key Features

- **Web Development**: Includes technologies like Servlets and JSP (JavaServer Pages) for dynamic content generation.
- **Enterprise APIs**: Offers frameworks like JPA (Java Persistence API) for database interaction, JMS (Java Messaging Service) for messaging, and EJB (Enterprise JavaBeans) for implementing business logic.
- **Scalability**: Built-in support for clustering, load balancing, and session management.
- **Security**: Enterprise-grade features like role-based access control and encryption.
- **Integration**: Facilitates integration with external systems via web services and REST APIs.

Applications

- High-volume e-commerce platforms.

- Financial systems handling secure transactions.
- Backend systems for large-scale web applications.

Examples

- Online booking systems.
- Inventory management tools for enterprises.

---

3. Java Micro Edition (Java ME)

Overview

Java Micro Edition (Java ME) is a lightweight and specialized version of Java designed for devices with limited resources, such as embedded systems, feature phones, and Internet of Things (IoT) devices. It provides a compact runtime environment and a subset of Java SE functionalities, optimized for low-memory and low-power environments.

Key Features

- **Configurations**: Divided into CLDC (Connected Limited Device Configuration) for basic devices and CDC (Connected Device Configuration) for more advanced systems.
- **Profiles**: Provides frameworks for specific device requirements, such as the Mobile Information Device Profile (MIDP) for mobile phones.
- **Hardware Interaction**: Includes APIs for working with hardware like sensors, displays, and connectivity modules.
- **Portability**: Ensures applications can run on a wide range of devices with minimal modification.

Applications

- Mobile apps for feature phones.
- Embedded software in appliances and vehicles.
- IoT solutions like smart home systems and wearable devices.

Examples

- Early mobile games and tools.
- Control software for embedded systems like thermostats or smart meters.

---

4. JavaFX

Overview

JavaFX is a framework focused on building modern, visually rich applications with sophisticated user interfaces. It was introduced as a replacement for older GUI frameworks like Swing and AWT. JavaFX simplifies the development of desktop applications with features like CSS styling, multimedia support, and advanced animation capabilities.

Key Features

- **Rich UI Components**: Offers pre-built controls like buttons, charts, and tables.
- **Multimedia Integration**: Supports video and audio playback, enabling multimedia-rich applications.
- **2D/3D Graphics**: Provides APIs for rendering complex graphics and animations.
- **FXML**: Allows developers to define UI elements using an XML-based scripting language.
- **Web Content Integration**: Embeds web pages and dynamic content through WebView.

Applications

- Desktop applications with interactive user interfaces.
- Visualization tools for data analysis and reporting.
- Media-centric tools like video players and graphic design software.

Examples

- Dashboards with real-time analytics.
- Graphic design applications with animation capabilities.

Summary of Java Editions

Edition	**Purpose**	**Applications**	**Examples**
Java SE	Core functionality for general-purpose development	Standalone and desktop applications	IDEs, utilities, and productivity tools
Java EE	Enterprise-level development for scalable systems	E-commerce, financial apps, enterprise tools	Banking systems, online portals
Java ME	Lightweight Java for constrained devices	Embedded systems, IoT, and feature phones	Smart devices, early mobile games
JavaFX	Rich client applications with advanced UIs	Interactive desktop tools and media software	Data dashboards, graphic tools

Java's editions are designed to meet the diverse needs of developers, making it a powerful platform for building applications across a wide range of devices and industries.

JAVA VERSIONS AND ITS IMPORTANT UPDATES

1. **JDK 1.0 (January 1996)**

    a. **Key Features**: Initial release; introduced core features like the Java language, JVM, applets, and basic APIs.

2. **JDK 1.1 (February 1997)**
    a. **Key Features**: Added inner classes, JavaBeans, JDBC (database connectivity), and improved event handling.
3. **J2SE 1.2 (December 1998)**
    a. **Key Features**: Introduced Swing for GUI, the Collections Framework, and Java Plug-in for web integration.
4. **J2SE 1.3 (May 2000)**
    a. **Key Features**: Enhanced RMI, introduced JavaSound, and improved scalability with CORBA support.
5. **J2SE 1.4 (February 2002)**
    a. **Key Features**: Added assert keyword, NIO (New Input/Output), logging API, and XML processing capabilities.
6. **J2SE 5.0 (September 2004)**
    a. **Key Features**: Introduced generics, annotations, enumerations, varargs, and enhanced for-loop syntax.
7. **Java SE 6 (December 2006)**
    a. **Key Features**: Added scripting support (JavaScript engine), enhanced JDBC, and included web services API.
8. **Java SE 7 (July 2011)**
    a. **Key Features**: Introduced try-with-resources, the diamond operator (<>), and improved file handling (NIO.2).
9. **Java SE 8 (March 2014)**
    a. **Key Features**: Lambda expressions, Stream API, a new Date-Time API, and Optional for handling null values.
10. **Java SE 9 (September 2017)**
    a. **Key Features**: Introduced the Java Platform Module System (JPMS), JShell (interactive REPL), and HTTP/2 client.
11. **Java SE 10 (March 2018)**

   a. **Key Features**: Local variable type inference with the var keyword and minor performance enhancements.

12. **Java SE 11 (September 2018)**
    a. **Key Features**: Long-Term Support (LTS) release; added the HTTP Client API and removed JavaFX from the core JDK.
13. **Java SE 12 (March 2019)**
    a. **Key Features**: Previewed switch expressions and introduced garbage collector enhancements.
14. **Java SE 13 (September 2019)**
    a. **Key Features**: Text blocks (multi-line strings) and enhanced garbage collection.
15. **Java SE 14 (March 2020)**
    a. **Key Features**: Records (preview feature) and improved pattern matching for instance of.
16. **Java SE 15 (September 2020)**
    a. **Key Features**: Sealed classes (preview feature) and finalized text blocks.
17. **Java SE 16 (March 2021)**
    a. **Key Features**: Finalized records and pattern matching for instanceof.
18. **Java SE 17 (September 2021)**
    a. **Key Features**: Long-Term Support (LTS) release; introduced sealed classes and improved security APIs.
19. **Java SE 18 (March 2022)**
    a. **Key Features**: UTF-8 as the default character set and previewed simple web server.
20. **Java SE 19 (September 2022)**
    a. **Key Features**: Introduced virtual threads (preview) and structured concurrency (preview).
21. **Java SE 20 (March 2023)**
    a. **Key Features**: Updated pattern matching, virtual threads, and structured concurrency.
22. **Java SE 21 (September 2023)**

a. **Key Features**: Long-Term Support (LTS) release; finalized virtual threads and enhanced pattern matching.

23. Java SE 22 (March 2024)

a. **Key Features**: Introduces minor improvements to garbage collection and dynamic runtime optimizations.

24. **Java SE 23 (September 2024)**

a. **Key Features**: Anticipated advancements in modularity and refinements to the foreign function and memory API.

JDK,JRE,JVM

1. JDK (Java Development Kit)

The **Java Development Kit (JDK)** is a complete software development package provided by Oracle for building Java applications. It contains a set of development tools, libraries, and frameworks required to create, compile, and run Java programs. The key component of the JDK is the **Java Compiler** (javac), which converts human-readable source code into **bytecode**, a machine-independent code that the Java Virtual Machine (JVM) can execute. The JDK also includes the **Java Runtime Environment (JRE)**, which provides the necessary runtime libraries and the JVM itself for executing Java programs. Additionally, the JDK includes debugging tools (like **jdb**), utilities for packaging Java applications into **JAR (Java Archive)** files, and documentation to assist developers in writing code efficiently. Developers rely on the JDK to create everything from simple console applications to large-scale enterprise software, and it is essential for anyone who writes or compiles Java code.

2. JVM (Java Virtual Machine)

The **Java Virtual Machine (JVM)** is an abstract computing machine that enables a computer to run Java programs. It is an essential component of the Java platform that provides the environment in which Java bytecode can be executed, regardless of the underlying hardware and operating system. The JVM performs crucial tasks such as interpreting or compiling bytecode into native machine code, managing memory (using a garbage collector to automatically reclaim memory), and providing platform independence. This means that Java programs can run on any device or operating system with a compatible JVM installed, following the "write once, run anywhere" philosophy. The JVM has three main parts: the **class loader**, which loads the class files; the **execution engine**, which runs the bytecode; and the **garbage collector**, which handles automatic memory management. It also provides an exception-handling mechanism and other vital services like multithreading and synchronization to help optimize Java applications at runtime.

3. JRE (Java Runtime Environment)

The **Java Runtime Environment (JRE)** is the environment in which Java applications are executed. It includes the **JVM**, which is responsible for running Java bytecode, and the necessary libraries and files to support the application at runtime. The JRE provides essential components such as class libraries, which consist of Java API packages that offer functionality like I/O operations, networking, database access, and more. However, the JRE does not include development tools like the Java compiler (javac), which are essential for building Java applications, making it distinct from the JDK. The primary function of the JRE is to provide a runtime environment for users to run Java applications without needing to install the full development environment. It is typically used by individuals who only want to run Java programs, such as end-users or systems administrators. Without the JRE, a user cannot run a Java application because it lacks the necessary components to execute the bytecode produced by the JDK.

FEATURES OF JAVA

1. Simple

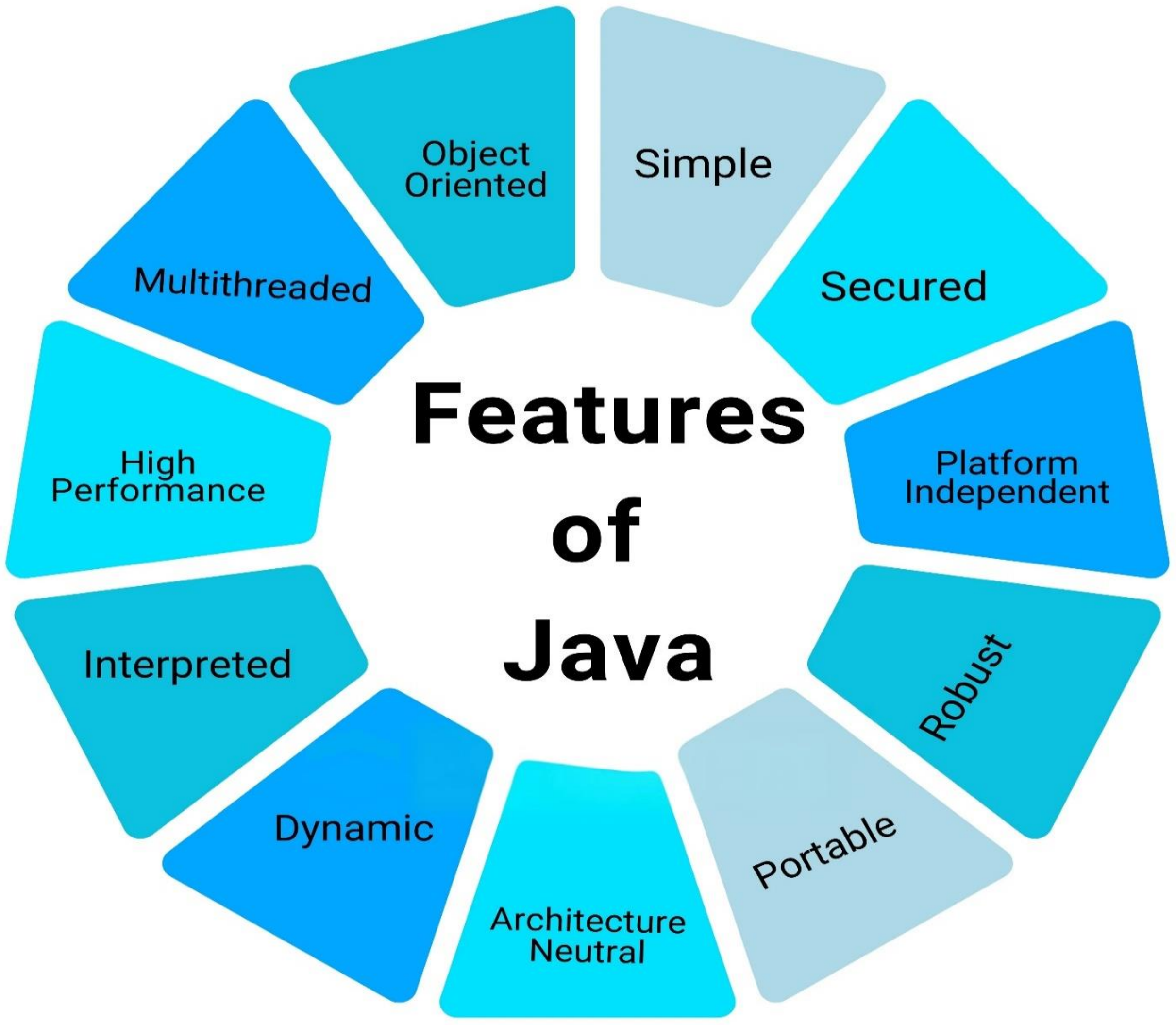
Java was designed with simplicity in mind. One of its main objectives was to create a language that was easy to learn, use, and maintain. It simplifies many features that other programming languages (like C and C++) struggle with, such as **pointer arithmetic**, **manual memory management**, and **complex syntax**. Java does not allow direct access to memory, which significantly reduces the risk of errors like memory corruption. Additionally, Java's clear syntax, which is influenced by C, makes it easier for developers to pick up and build applications without facing steep learning curves.

2. Object-Oriented

Java is an **object-oriented programming (OOP)** language, and its design revolves around the use of objects. This means that Java organizes code into classes, which represent objects, and each class defines the properties and behaviors (methods) that an object can have. The key features of OOP in Java—**encapsulation**, **inheritance**, **polymorphism**, and **abstraction**—enable code reuse, modularization, and maintainability. These principles help developers structure software in a more intuitive, scalable, and flexible way. For example, by inheriting common behavior from a base class, a subclass can extend functionality without duplicating code.

3. Portable

Java is **portable** because it follows the principle of "Write Once, Run Anywhere." Unlike other programming languages, Java doesn't compile directly into machine code specific to a hardware architecture. Instead, it compiles into **bytecode**, which can be run on any machine that has a **Java Virtual Machine (JVM)** installed. This allows Java programs to execute seamlessly across different operating systems like Windows, macOS, and Linux without needing any modifications. The ability to execute the same code across various environments without recompilation is one of Java's strongest features for enterprise applications.



4. Platform Independent

One of Java's standout qualities is its **platform independence**. When Java code is compiled, it is converted into an intermediate form known as **bytecode**. This bytecode is not dependent on any specific operating system or hardware architecture. When you run the Java application, the **JVM** interprets the bytecode and converts it into machine-specific instructions based on the platform it is running on. This ability to execute on any platform with a JVM allows Java to be used in diverse environments, from mobile devices to enterprise servers, making it a highly adaptable language for modern applications.

5. Secured

Security is a fundamental design feature of Java. Java applications are **sandboxed**, meaning that they are executed within a secure environment where the JVM monitors and limits the operations they can perform. This prevents unauthorized access to system resources or data, offering a robust level of security. Java also includes **bytecode verification** to ensure that only verified and safe bytecode is allowed to run. Additionally, Java has extensive libraries for **encryption**, **authentication**, **access control**, and **secure communication**, which are essential for building secure networked applications, such as web applications and financial systems.

6. Robust

Java is considered a **robust** language because it incorporates a variety of features to ensure reliable execution. One of these features is **automatic garbage collection**, which prevents memory leaks by automatically reclaiming memory that is no longer in use. Additionally, Java's **strong type checking** and exception handling mechanism ensure that programs are less prone to errors. The language also enforces strict **null-pointer checks** and **bounds checking** to avoid crashes or unexpected behavior. These features make Java applications more stable and less likely to fail during execution.

7. Architecture Neutral

Java is **architecture neutral** because the bytecode generated by the Java compiler is not tied to any specific processor architecture. Unlike languages that generate machine-specific code, Java bytecode is designed to be executed on any machine with a compatible JVM. Whether the host system is based on an **Intel processor** or an **ARM processor**, Java bytecode remains consistent. This architecture-neutral approach means developers don't have to worry about hardware-specific optimizations, and Java programs can seamlessly run across various platforms, from desktops to cloud-based servers.

8. Interpreted

Java is an **interpreted** language, which means its bytecode is not directly executed by the operating system's native processor. Instead, the **Java Virtual Machine (JVM)** interprets the bytecode and converts it into machine-specific instructions during runtime. This layer of interpretation allows for **portability** since the JVM can be implemented on any platform. Java also employs **Just-In-Time (JIT) compilation**, where frequently executed bytecode is compiled into native machine code at runtime for better performance. JIT compilation allows Java to strike a balance between the flexibility of interpretation and the performance of native code execution.

9. High Performance

While Java is an interpreted language, it achieves **high performance** through several optimizations. The **Just-In-Time (JIT)** compiler converts frequently executed bytecode into machine-specific code, improving the execution speed. Java's **multithreading** capabilities allow for efficient parallel processing, making it ideal for high-performance applications that need to handle multiple tasks concurrently, such as real-time data processing or gaming. Additionally, Java's efficient memory management and **garbage collection** process help keep performance high by ensuring that unused objects are promptly removed from memory, reducing the chance of performance degradation over time.

10. Multithreaded

Java's **multithreading** capability enables the concurrent execution of multiple threads (independent units of execution) within a program. This feature is critical for applications that need to perform multiple tasks at the same time, such as processing data while simultaneously responding to user input or performing background tasks. Java provides built-in support for multithreading through the **Thread class** and **Runnable interface**, which makes it easier for developers to manage threads without dealing with the complexity of low-level thread management. Java's ability to handle multithreading efficiently makes it well-suited for developing high-performance and interactive applications, such as online games, media players, and real-time applications.

11. Distributed

Java has extensive support for **distributed computing**, allowing developers to create applications that run on multiple machines across a network. Java provides powerful tools for building distributed systems, such as **Remote Method Invocation (RMI)**, which allows objects to communicate with each other over a network as if they were local. Other features, like **Java Message Service (JMS)** and **Sockets**, help with message-based communication, enabling seamless data exchange between distributed applications. Java's built-in networking capabilities make it a great choice for developing web applications, cloud-based systems, and enterprise solutions that require distributed processing and communication.

12. Dynamic

Java is a **dynamic** language, which means it can adapt to changing conditions at runtime. For instance, Java allows for **dynamic class loading**, where classes are loaded into memory only when needed, which makes the application more flexible. Java's **reflection API** enables runtime inspection and modification of classes, methods, and fields, allowing developers to build more dynamic and extensible applications. This dynamic capability is also essential in scenarios where Java applications need to interact with external libraries or plugins that can be added or updated without restarting the application.

KEYWORDS IN JAVA

In Java, keywords are reserved words that have a predefined meaning and serve a specific function in the language. They are fundamental to Java's syntax and structure and cannot be used as identifiers (such as names for variables, methods, or classes). Keywords help define the behavior, flow, and structure of a Java program. They form the core building blocks that allow the programmer to define logic, control the program flow, and interact with objects. Java has a set of 50 keywords that the Java compiler recognizes and interprets in a specific way. These keywords cover various aspects of Java, including class declarations, control flow, error handling, access control, and object-oriented concepts.

- **class**: Defines a class, which is a template for creating objects and defining methods and attributes.
- **public**: Specifies that a member (method, variable) is accessible from any other class.
- **private**: Specifies that a member (method, variable) is accessible only within its own class.
- **protected**: Specifies that a member is accessible within its own package and by subclasses.
- **static**: Defines a member (method or variable) that belongs to the class rather than instances of the class.
- **final**: Used to declare constants, prevent method overriding, or prevent inheritance of a class.
- **void**: Specifies that a method does not return any value.
- **if**: Defines a conditional statement to execute code based on a boolean expression.
- **else**: Specifies the alternative block of code in an if-else statement.
- **for**: Defines a loop that executes a block of code a specified number of times.
- **return**: Used to return a value from a method or to exit from a method.
- **try**: Defines a block of code that may throw an exception, used in conjunction with catch and finally.

- **catch**: Used to handle exceptions that occur in a try block.
- **import**: Imports other classes or packages into the current class to use them.
- **new**: Used to create new objects or arrays in Java.

IDENTIFIERS IN JAVA

In Java, **identifiers** are names used to refer to various program elements such as variables, methods, classes, and objects. They play a crucial role in allowing developers to reference and manipulate data throughout the program. Java has specific rules for naming identifiers: they must begin with a letter (a-z, A-Z), an underscore (_), or a dollar sign ($), followed by any combination of letters, digits (0-9), underscores, or dollar signs. Importantly, Java is case-sensitive, so identifiers like myVariable and MyVariable would be treated as distinct. Identifiers cannot be the same as reserved keywords in Java, such as if, for, or class. While identifiers can be of any length, it’s best practice to use meaningful names that describe the purpose of the variable or method to make the code more readable. Common naming conventions include using camel case (e.g., totalAmount) for variables and methods, and Pascal case (e.g., StudentDetails) for class names. Avoid using digits at the beginning of identifiers, and refrain from using special characters other than underscores and dollar signs.

Examples:-

❖ **Valid Identifiers**:

- totalAmount
- customerName
- calculateSalary
- EmployeeDetails
- _value
- $amount

- itemCount

❖ **Invalid Identifiers**:

- 123abc (starts with a digit)
- if (reserved keyword)
- int (reserved keyword)
- my-variable (contains a hyphen, which is not allowed)

PACKAGES IN JAVA

In Java, a **package** is a namespace that organizes a set of related classes, interfaces, and sub-packages. It serves as a way to group classes logically, preventing naming conflicts and enhancing the organization of code. Packages are essential in managing large-scale Java applications, ensuring that the code is modular, easily maintainable, and accessible by other programs.

Uses of Packages:

1. **Name Collision Prevention**: By grouping classes into different packages, you can avoid conflicts between classes with the same name but different functionalities. For example, two packages can both have a Date class, but they will not conflict if they are in different namespaces (like java.util.Date and java.sql.Date).
2. **Access Control**: Packages play a key role in access control. Java uses access modifiers like public, protected, and private to control the visibility of classes and their members. Classes in the same package can access each other's members with default (package-private) access or protected access.
3. **Code Organization**: Packages help in organizing code by logically grouping classes that share common functionality, making it easier to maintain and navigate. This is particularly useful in large projects.

4. **Reusability**: When classes are packaged together, they can be reused in different parts of the same project or in other projects, enhancing modularity and reducing redundancy.

Types of Packages:

1. **Built-in Packages**: These are pre-defined packages provided by the Java Development Kit (JDK) and contain a vast collection of classes that help with common tasks such as data manipulation, networking, file handling, etc. Some commonly used built-in packages include:
   - **java.lang**: Contains fundamental classes, such as String, Math, System, and Object, which are automatically imported into every Java program.
   - **java.util**: Provides utility classes like ArrayList, HashMap, Date, Scanner, and others for data structures and algorithms.
   - **java.io**: Contains classes for input and output (I/O) operations, such as reading and writing to files.
   - **java.net**: Provides classes for networking functionality, such as Socket, URL, and HttpURLConnection.
   - **java.sql**: Includes classes for database connectivity and manipulation using JDBC (Java Database Connectivity).

DATA TYPES IN JAVA

In Java, a **data type** is a fundamental aspect of the language, defining the type and range of values a variable can hold and the operations that can be performed on it. Data types help allocate the appropriate amount of memory for storing data and ensure that operations on variables are valid and predictable. This strict classification contributes to Java’s reputation as a statically-typed and type-safe language, where the type of a variable must be declared at the time of its creation. For instance, specifying int for an integer value ensures that only whole numbers can be stored, while a boolean is restricted to true or false.

Data types are crucial in enforcing program stability and preventing errors such as assigning invalid data to a variable. They allow the compiler to detect type mismatches during compile time rather than at runtime, significantly reducing potential bugs. Java offers a diverse range of data types, categorized into **primitive** and **non-primitive types**, allowing developers to store simple data (like numbers and characters) as well as complex data (like arrays, objects, and strings).

PRIMITIVE DATA TYPES

In Java, primitive data types are the most fundamental building blocks of data storage and computation. They are predefined by the language and represent simple values such as integers, floating-point numbers, characters, and boolean logic. Unlike objects, primitive data types store actual values directly in memory and are not associated with methods or additional properties, making them efficient and lightweight. These types are used for core functionalities, such as performing mathematical operations, storing state variables, and handling logical decisions in programs. While Java is predominantly object-oriented, its use of primitive data types is one reason it is not considered a 100% object-oriented language. Primitive types are not objects; they cannot inherit from classes or participate in the object hierarchy. This deliberate design choice ensures better performance and simpler handling of basic data, balancing object-oriented principles with practicality and efficiency.

TYPES OF PRIMITIVE DATA TYPES AVAILABLE IN JAVA

Type	Size	Range	Example Usage
byte	1 byte	-128 to 127	Memory-efficient storage of small numbers
short	2 bytes	-32,768 to 32,767	For numbers larger than byte but smaller than int
int	4 bytes	-2,147,483,648 to 2,147,483,647	Default for integer numbers
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	For very large numbers
float	4 bytes	Approx. ±3.4e–038 to ±3.4e+038	Single-precision floating-point
double	8 bytes	Approx. ±1.7e–308 to ±1.7e+308	Default for decimal numbers
char	2 bytes	0 to 65,535 (Unicode values)	Single character storage
boolean	1 bit	true or false	For logic or decision-making

BYTE:

In Java, **byte** is one of the eight primitive data types. It is used to represent small integer values. A byte occupies **1 byte (8 bits)** of memory and can store values in the range of **-128 to 127**. This makes it particularly useful for situations where memory efficiency is crucial and the range of values is limited to these small integers.

EXAMPLE PROGRAM :-

```
public class ByteProgram {

        public static void main(String[] args) {

            byte num1 = 50;

            byte num2 = -30;

            System.out.println("The value of num1 is: " + num1);

            System.out.println("The value of num2 is: " + num2);

        } }
```

OUTPUT:

```
The value of num1 is: 50
The value of num2 is: -30
```

SHORT

In Java, **short** is one of the primitive data types used to store small integer values. It is larger than a byte but smaller than an int, making it suitable for situations where memory usage is a concern, but a byte is too small to handle the required range of values.

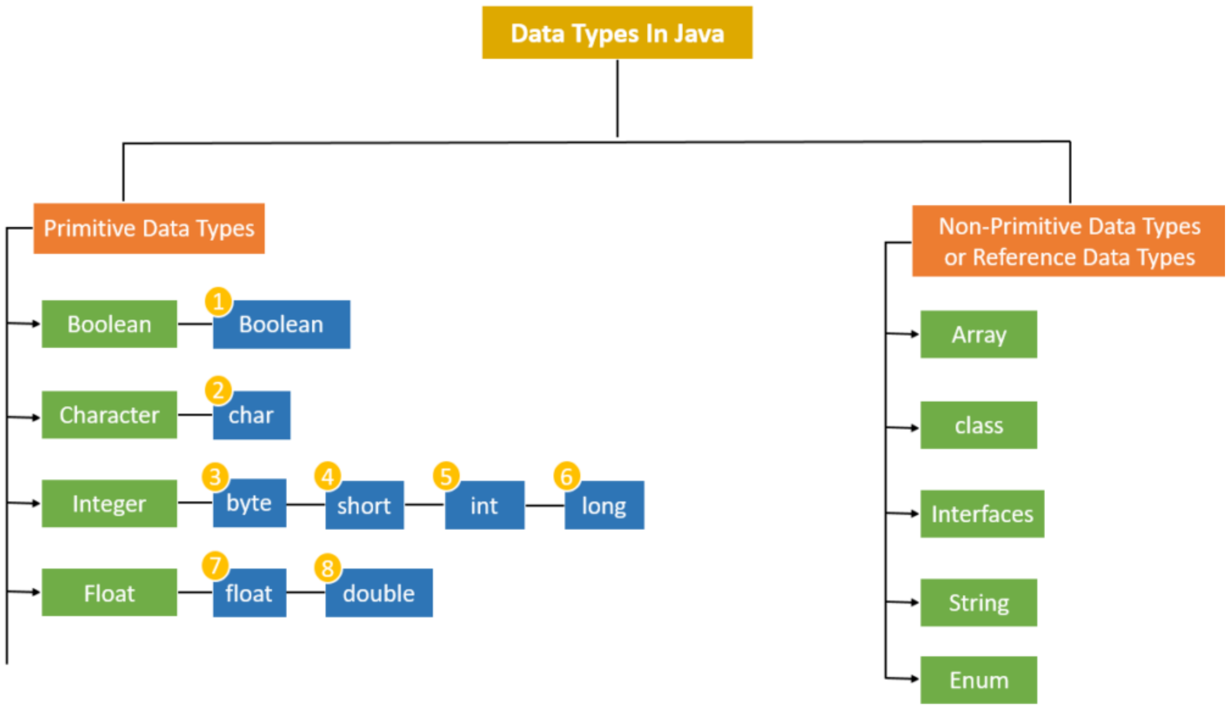
EXAMPLE PROGRAM:-

```
public class ShortProgram {

      public static void main(String[] args) {

             short maxShort = 32767;

             short overflowedValue = (short) (maxShort + 1);

             System.out.println("Maximum short value: " + maxShort);
```

```
                System.out.println("After adding 1 (Overflow): " + overflowedValue);
}}
```

OUTPUT:-

```
Maximum short value: 32767
After adding 1 (Overflow): -32768
```



INT

In Java, **int** is a primitive data type used to store whole numbers (integers). It is one of the most commonly used data types because it provides a balance between memory efficiency and a wide range of values. The int data type is ideal for general-purpose numerical operations where the range of values is expected to fit within its limits.

EXAMPLE PROGRAM:-

```
public class IntProgram {

    public static void main(String[] args) {

            int num1 = 15;

            int num2 = 7;

            int sum = num1 + num2;

            int difference = num1 - num2;

            int product = num1 * num2;

            int quotient = num1 / num2;

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

            System.out.println("Difference: " + difference);

            System.out.println("Product: " + product);

            System.out.println("Quotient: " + quotient);

        }

    }
```

OUTPUT:-

```
Sum: 22
Difference: 8
```

```
Product: 105
Quotient: 2
```

LONG

In Java, **long** is a primitive data type used to store large integer values. It is an extended version of int with a much larger range, making it suitable for applications that require handling very large numerical values beyond the capacity of int.

EXAMPLE PROGRAM:-

```
public class LongProgram {

public static void main(String[] args) {

long population = 7830000000L;

     long distanceToMoon = 384400000L;

     System.out.println("World Population: " + population);

     System.out.println("Distance to Moon: " + distanceToMoon + " meters");

}

}
```

OUTPUT:-

```
World Population: 7830000000
Distance to Moon: 384400000 meters
```

FLOAT

In Java, the **float** data type is a single-precision, 32-bit IEEE 754 floating-point number used to store decimal values. It is designed for scenarios where memory efficiency is more important than precision. Unlike integer types, float can handle fractional values, making it suitable for measurements, approximations, and calculations where exact precision is not critical.

EXAMPLE PROGRAM:-

```
public class FloatProgram {

      public static void main(String[] args) {

      float num1 = 5.5f;

     float num2 = 10.2f;

     if (num1 > 0 && num2 > 0) {

        System.out.println("Both numbers are positive.");

     }

     if (num1 < 0 || num2 > 0) {

        System.out.println("At least one number is positive.");

     }

     boolean isNum1Negative = !(num1 > 0);

     System.out.println("Is num1 negative? " + isNum1Negative);

     if ((num1 > 5.0f && num2 < 15.0f) || num1 == 5.5f) {

        System.out.println("Combined logical check passed.");

     }

}}
```

OUTPUT:-

```
Both numbers are positive.
At least one number is positive.
Is num1 negative? false
Combined logical check passed.
```

DOUBLE

In Java, **double** is a primitive data type used to represent floating-point numbers with **double precision**. It is part of the IEEE 754 standard for floating-point arithmetic, which specifies how floating-point numbers are stored and manipulated. The double data type is used to store decimal values where higher precision is required, and it allows for more accurate representation of numbers than its counterpart, the float.

EXAMPLE PROGRAM:-

```
public class DoubleProgram {
                              public static void main(String[] args) {
                                      double pi = 3.141592653589793;
      double radius = 7.5;
      double area = pi * radius * radius;
      System.out.println("Area of the circle: " + area);
      double largeNumber = 1.234567890123456e15;
      System.out.println("Large number: " + largeNumber);
      double smallNumber = 1.234567890123456e-15;
      System.out.println("Small number: " + smallNumber);
}}
```

OUTPUT:-

```
Area of the circle: 176.71458676442586
Large number: 1.234567890123456E15
Small number: 1.234567890123456E-15
```

CHAR

In Java, **char** is a primitive data type that is used to store a single character. Internally, it is represented as a **16-bit unsigned integer**. This means that each char value occupies **2 bytes (16 bits)** in memory. The reason for this is that Java uses the **Unicode** character set, which can represent a wide range of characters, symbols, and even characters from different languages, extending beyond the basic ASCII character set.

EXAMPLE PROGRAM:-

```
public class CharProgram {
                    public static void main(String[] args) {
                    char letter = 'A';
    int unicodeValue = letter;
    System.out.println("Character: " + letter);
    System.out.println("Unicode Value (Internal Storage): " + unicodeValue);
                    }
}
```

OUTPUT:-

```
Character: A
Unicode Value (Internal Storage): 65
```

BOOLEAN

In Java, the **boolean** data type is one of the eight primitive data types and is primarily used to store **binary values**. A boolean can only have two possible values: **true** or **false**. These two values represent the fundamental concept of logical truth, which is essential in decision-making and flow control within a program.

EXAMPLE PROGRAM:-

```
public class BooleanProgram {

public static void main(String[] args) {

int age = 20;

        boolean hasPermission = true;

        if (age >= 18 && hasPermission) {

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

        } else {

            System.out.println("Access denied.");

        }

        boolean isEligibleForDiscount = !(age < 18);

        System.out.println("Eligible for discount: " + isEligibleForDiscount);

                                }}
```

OUTPUT:-

```
Access granted.
Eligible for discount: true
```

STRINGS IN JAVA

In Java, a **String** is a sequence of characters used to represent textual data. Strings are objects of the String class in the java.lang package and are among the most commonly used data types in Java programming. Strings in Java are **immutable**, meaning once created, their content cannot be changed. This immutability provides security, makes them thread-safe, and optimizes memory usage by leveraging a feature called the **string pool**. Strings can be declared and used in various ways, with each approach affecting how they are stored in memory. The two primary types of strings in Java are **String Literal** and **String Object**.

TYPES OF STRINGS IN JAVA

Strings in Java can be created and managed using two primary approaches they are:

1. String Literal

2. String Object

1.STRING LITERALS

A string literal is created when you assign a value directly to a String variable using double quotes, This approach stores the string in a special memory area called the **string constant pool**, which resides in the heap memory. The constant pool optimizes memory usage by avoiding duplicate strings. If a string with the same value already exists in the pool, the reference to that string is reused instead of creating a new one. This mechanism makes string literals efficient for frequently used strings.

SYNTAX:

```
String str = "Hello";
```

Key Characteristics of String Literals:

- Managed by the **string pool**.
- Reference is reused if the value already exists in the pool.
- Memory-efficient for common strings.

<u>EXAMPLE PROGRAM:</u>

```
public class LiteralProgram { public static void main(String[] args) {

String literal1 = "Java";

    String literal2 = "Java";

    String object1 = new String("Java");

    String object2 = new String("Java");

    System.out.println("Comparing String Literals:");

    System.out.println("literal1 == literal2: " + (literal1 == literal2));

    System.out.println("literal1.equals(literal2): " + literal1.equals(literal2));

    System.out.println("\nComparing String Objects:");

    System.out.println("object1 == object2: " + (object1 == object2));

    System.out.println("object1.equals(object2): " + object1.equals(object2));

    System.out.println("\nCross-Comparing Literal and Object:");

    System.out.println("literal1 == object1: " + (literal1 == object1));

    System.out.println("literal1.equals(object1): " + literal1.equals(object1));

    System.out.println("\nMemory Reference Details:");
```

```
        System.out.println("literal1 hashcode: " + System.identityHashCode(literal1));

        System.out.println("literal2 hashcode: " + System.identityHashCode(literal2));

        System.out.println("object1 hashcode: " + System.identityHashCode(object1));

        System.out.println("object2 hashcode: " + System.identityHashCode(object2));

                    }}
```

OUTPUT:-

```
Comparing String Literals:
literal1 == literal2: true
literal1.equals(literal2): true
Comparing String Objects:
object1 == object2: false
object1.equals(object2): true
Cross-Comparing Literal and Object:
literal1 == object1: false
literal1.equals(object1): true
Memory Reference Details:
literal1 hashcode: 2018699554
literal2 hashcode: 2018699554
object1 hashcode: 1311053135
object2 hashcode: 118352462
```

2.STRING OBJECT

A string object is created using the new keyword, This approach explicitly creates a new object in the **heap memory**, bypassing the string pool. Even if an identical string exists in the pool, a new object is created. While this provides flexibility, it can be less memory-efficient since it does not leverage the reuse feature of the string pool.

Key Characteristics of String Objects:

- Always creates a **new object** in the heap.
- Does not reuse references from the string pool.
- Useful when modifications to the string content are anticipated (though for actual mutability, StringBuilder or StringBuffer is preferred).

EXAMPLE PROGRAM

```
public class ObjectStringProgram {

public static void main(String[] args) {

String original = "Java";

String modified = original + " Programming";

String appended = original + " Learning";

System.out.println("Original String: " + original);

System.out.println("Modified String: " + modified);

System.out.println("Appended String: " + appended);

System.out.println("Original hashcode: " + System.identityHashCode(original));

System.out.println("Modified hashcode: " + System.identityHashCode(modified));

System.out.println("Appended hashcode: " + System.identityHashCode(appended));

}}
```

OUTPUT:-

```
Original String: Java
Modified String: Java Programming
Appended String: Java Learning
```

```
Original hashcode: 2018699554
Modified hashcode: 1311053135
Appended hashcode: 118352462
```

HEAP MEMORY AND OBJECT MEMORY IN JAVA

Java's memory management system is designed to ensure efficient allocation, usage, and reclamation of memory, which is critical for applications. Among its memory areas, **heap memory** plays a central role in storing objects, including their associated data and metadata.

HEAP MEMORY IN JAVA

Heap memory is a runtime memory area that the Java Virtual Machine (JVM) uses to allocate memory for **all Java objects** and **class instances**. It is part of the JVM's larger memory model and is shared across all threads in a Java application. Heap memory is dynamic, meaning objects are created and destroyed during program execution.

OBJECT MEMORY IN JAVA

In Java, when an object is created, it resides in the **heap memory**, which is dynamically allocated at runtime. Each object in the heap is structured in a way that includes both its data and additional information used by the JVM. This detailed explanation delves into the components, memory layout, and lifecycle of an object in heap memory.

STRING METHODS

- length()
- charAt(int index)
- substring(int beginIndex, int endIndex)
- toLowerCase()
- toUpperCase()

- trim()
- replace(char oldChar, char newChar)
- equals(Object anotherString)
- equalsIgnoreCase(String anotherString)
- contains(CharSequence sequence)

Length()

The length() method is one of the most frequently used methods in the String class in Java. It returns the **number of characters** present in the string, including spaces, punctuation, and any other characters. This method does not require any arguments, and it returns an **integer** value representing the total length of the string.

Key Points:

- The method returns the **total number of characters**, including whitespace and special characters.
- It is a non-static method, meaning it can be called only on an instance of the String class.
- It does not count **null** or **empty** strings, but rather the actual content within a String.

EXAMPLE PROGRAM

```
public class MethodLengthProgram {

public static void main(String[] args) {

String str1 = "Hello, Java!";

String str2 = "Java";

System.out.println("Length of str1: " + str1.length());

System.out.println("Length of str2: " + str2.length());
}

}
```

OUTPUT:-

```
Length of str1: 12
Length of str2: 4
```

charAt()

The charAt(int index) method is a fundamental method in the String class of Java. It is used to retrieve a **single character** from a string at the specified position (index). The index is zero-based, meaning the first character of the string is at index 0, the second at index 1, and so on. This method returns a **char** value, representing the character at the given index.

Key Points:

- **Returns a char**: The method returns a single character (char) at the specified index.
- **Indexing starts from 0**: In Java strings, indexing starts at 0, so the first character is at position 0, the second at position 1, and so on.
- **Throws StringIndexOutOfBoundsException**: If the provided index is negative or greater than or equal to the length of the string, it throws an exception.

EXAMPLE PROGRAM

```
public class MethodCharAtProgram {

public static void main(String[] args) {

String str = "Hello, Java!";

char characterAtIndex3 = str.charAt(3);

System.out.println("Character at index 3: " + characterAtIndex3);
}

}
```

OUTPUT:-

```
Character at index 3: l
```

substring()

The substring(int beginIndex, int endIndex) method is part of the String class in Java and is used to extract a portion of a string from a given starting index (beginIndex) to an ending index (endIndex). This method is very useful when you want to work with a specific section of a string, such as extracting a word, sentence, or part of a path.

Key Points:

- **Start Index (beginIndex)**: The index where the substring begins, inclusive. It is the first position of the substring in the original string.
- **End Index (endIndex)**: The index where the substring ends, exclusive. The character at this position is **not** included in the resulting substring.
- **Zero-based Index**: Both beginIndex and endIndex are zero-based. The first character in the string has index 0, the second character has index 1, and so on.
- **StringIndexOutOfBoundsException**: If the indices are out of range (negative or greater than the string length), it throws a StringIndexOutOfBoundsException.

EXAMPLE PROGRAM

```
public class ProgramSubString {

public static void main(String[] args) {

String date = "2024-11-26";
String year = date.substring(0, 4);

String month = date.substring(5, 7);

String day = date.substring(8, 10);

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

System.out.println("Month: " + month);

System.out.println("Day: " + day);    }  }
```

OUTPUT:-

```
Year: 2024
Month: 11
Day: 26
```

toLowerCase()

The toLowerCase() method in Java is used to convert all the characters in a given string to lowercase. This method is part of the String class, which provides a wide range of useful methods for string manipulation. It is a very commonly used method, particularly for case-insensitive operations such as comparing strings, formatting input, or processing user data.

Key Points:

- **Converts All Characters to Lowercase**: The method converts every character in the string to its lowercase equivalent.
- **Locale Sensitivity**: The toLowerCase() method operates based on the default locale of the system. It may behave differently in different locales for some characters, such as I in Turkish, where it doesn't convert to i as expected in other languages.
- **Returns a New String**: The method does not modify the original string. Instead, it returns a new string with all characters in lowercase.

- **Does Not Affect Non-Alphabetic Characters**: Non-alphabetic characters such as numbers, spaces, and punctuation remain unchanged.

EXAMPLE PROGRAM

```
public class LowerCaseProgram {

public static void main(String[] args) {

String str = "Java Programming";

String lowerStr = str.toLowerCase();



System.out.println("Original String: " + str);

System.out.println("Lowercase String: " + lowerStr);

}

}
```

OUTPUT:-

```
Original String: Java Programming
Lowercase String: java programming
```

toUpperCase()

The toUpperCase() method in Java is a built-in method provided by the String class that converts all the characters in a given string to uppercase. This method is often used when you need to format text for uniform display, case-insensitive comparison, or when transforming user input for processing.

What Does toUpperCase() Do?

- **Converts Characters to Uppercase**: The toUpperCase() method transforms every lowercase letter in a string to its uppercase equivalent. For example, "hello world" becomes "HELLO WORLD".
- **Returns a New String**: Similar to toLowerCase(), the toUpperCase() method returns a new string and does not modify the original string.
- **Locale Sensitivity**: The method works based on the system's default locale, which can influence the result in certain languages. For example, in some languages, there are special rules for converting characters to uppercase.
- **Non-Alphabetic Characters Remain Unchanged**: The method only affects alphabetic characters. Non-alphabetic characters, such as numbers, spaces, and punctuation, remain unchanged.

EXAMPLE PROGRAM

```
public class UpperCaseProgram {

public static void main(String[] args) {

String message = "Welcome to Java Programming!";

String upperMessage = message.toUpperCase();

System.out.println("Original Message: " + message);

System.out.println("Uppercase Message: " + upperMessage);

}

}
```

OUTPUT:-

```
Original Message: Welcome to Java Programming!
```

```
Uppercase Message: WELCOME TO JAVA PROGRAMMING!
```

trim()

The trim() method in Java is a built-in method of the String class that removes leading and trailing whitespace from a string. This method is commonly used to clean up user input or text data that may have extra spaces at the beginning or end, which are usually unnecessary for comparison or display purposes.

- **Whitespace Removal**: The trim() method removes any space characters (such as spaces, tabs, and newline characters) from the beginning and end of a string, but does not affect the spaces in the middle of the string.
- **Does Not Modify the Original String**: Like other string methods, trim() does not change the original string because strings in Java are immutable. Instead, it returns a new string with the whitespace removed.

EXAMPLE PROGRAM

```
public class TrimProgram {

public static void main(String[] args) {

String[] usernames = {"  javaMaster  ", "  java  ", "developer ", "  javaMaster"};

String validUsername = "javaMaster";

for (String username : usernames) {

String trimmedUsername = username.trim();
if (trimmedUsername.equals(validUsername)) {

System.out.println("Valid username: " + trimmedUsername);

} else {

System.out.println("Invalid username: " + trimmedUsername);

}

}

}

}
```

OUTPUT;-

```
Valid username: javaMaster
Invalid username: java
Invalid username: developer
Valid username: javaMaster
```

replace(char oldChar, char newChar)

The replace() method in Java is a part of the String class, and it is used to replace occurrences of a specific character in a string with a new character. The method takes two arguments:

1. **oldChar**: The character in the string that needs to be replaced.
2. **newChar**: The character that will replace oldChar.

This method is useful for modifying strings, such as replacing certain characters with another character or adjusting text before processing or displaying it.

- **Does Not Modify the Original String**: Strings in Java are immutable, which means the replace() method does not change the original string. Instead, it returns a new string with the desired replacements.
- **Returns a New String**: The method returns a new string in which all occurrences of the specified character are replaced by the new character.

EXAMPLE PROGRAM

```
public class ReplaceProgram {

public static void main(String[] args) {

String sentence = "Java is an awesome programming language, but Java can be difficult!";

String sensitiveWord = "Java";

String replacement = "****";

String censoredSentence = sentence.replace(sensitiveWord, replacement);

System.out.println("Original Sentence: " + sentence);

System.out.println("Censored Sentence: " + censoredSentence);

}

}
```

OUTPUT:-

```
Original Sentence: Java is an awesome programming language, but Java can be difficult!
Censored Sentence: **** is an awesome programming language, but **** can be difficult!
```

equals(Object anotherString)

The equals(Object anotherString) method in Java is used to compare two strings for **equality**. It checks if the content of the string (the sequence of characters) is the same between two string objects, not just their reference memory location.

- **Returns a Boolean**: This method returns true if the two strings are exactly the same (case-sensitive), and false if they are not.
- **String Comparison**: Unlike ==, which compares memory references (addresses) of objects, the equals() method compares the actual characters in the strings.

EXAMPLE PROGRAM

```
public class EqualsProgram {

public static void main(String[] args) {

String str1 = "Java";

String str2 = "java";

boolean isEqual = str1.equals(str2);

System.out.println(isEqual);

}

}
```

OUTPUT:-

```
False
```

equalsIgnoreCase(String anotherString)

The equalsIgnoreCase(String anotherString) method in Java is a part of the String class, and it is used to compare two strings for **equality** while ignoring case differences. This means that the method will return true if the strings are the same, regardless of whether the characters are uppercase or lowercase.

Unlike the equals() method, which is **case-sensitive**, equalsIgnoreCase() is **case-insensitive**, making it useful when you need to compare strings without caring about case sensitivity.

EXAMPLE PROGRAM

```
public class IgnoreCaseProgram {

public static void main(String[] args) {

String str1 = "Java";

String str2 = "JAVA";

boolean isEqual = str1.equalsIgnoreCase(str2);

System.out.println(isEqual);

}

}
```

OUTPUT:-

```
True
```

contains(CharSequence sequence)

The contains(CharSequence sequence) method is part of the String class in Java, and it is used to check if a given sequence of characters exists within a string. This method returns a boolean value (true or false) depending on whether the specified sequence of characters (substring) is present in the string or not.

- **Case-Sensitive**: The contains() method is case-sensitive. This means that the characters must match exactly, including their case.
- **Returns true**: If the string contains the specified sequence.
- **Returns false**: If the string does not contain the specified sequence.

EXAMPLE PROGRAM

```
public class ContainsProgram {

public static void main(String[] args) {

String str = "Hello, welcome to Java programming!";

boolean containsJava = str.contains("Java");

System.out.println(containsJava);

}

}
```

OUTPUT:-

```
True
```

STRING BUILDER AND STRING BUFFER

1. Thread Safety

- **StringBuilder**: Not thread-safe (not synchronized).

- **StringBuffer**: Thread-safe (synchronized).
  Explanation: StringBuffer ensures thread safety by synchronizing its methods, while StringBuilder does not, making it faster in single-threaded applications.

2. Performance

- **StringBuilder**: Faster than StringBuffer as it is not synchronized.
- **StringBuffer**: Slower than StringBuilder due to synchronization.
  Explanation: The lack of synchronization in StringBuilder results in better performance compared to StringBuffer, which incurs additional overhead due to thread safety mechanisms.

3. Use Case

- **StringBuilder**: Preferred when thread-safety is not a concern.
- **StringBuffer**: Used when thread-safety is important (e.g., in multi-threaded environments).
  Explanation: StringBuffer is ideal for applications with multiple threads accessing the same object, while StringBuilder is better for single-threaded scenarios.

4. Synchronization

- **StringBuilder**: No synchronization mechanism.
- **StringBuffer**: Synchronization ensures thread safety for multiple threads accessing the object.
  Explanation: StringBuffer uses synchronization to avoid concurrent modification issues in multi-threaded environments, while StringBuilder skips this to improve performance when thread safety isn't required.

5. Constructor

- **StringBuilder**: StringBuilder() or StringBuilder(int capacity).
- **StringBuffer**: StringBuffer() or StringBuffer(int capacity). Explanation: Both classes offer constructors that allow you to specify an initial capacity for the underlying character array. However, StringBuffer adds synchronization features, while StringBuilder does not.

6. Method Behavior

- **StringBuilder**: All methods in StringBuilder are non-synchronized.
- **StringBuffer**: All methods in StringBuffer are synchronized. Explanation: The difference in synchronization affects how each class behaves in multi-threaded contexts. StringBuffer ensures that its methods are thread-safe, whereas StringBuilder prioritizes performance.

7. Memory Overhead

- **StringBuilder**: Lower memory overhead due to lack of synchronization.
- **StringBuffer**: Higher memory overhead due to synchronization. Explanation: Because StringBuffer synchronizes its methods, it consumes more memory compared to StringBuilder.

8. Mutability

- **StringBuilder**: Mutable, allows modification of content.
- **StringBuffer**: Mutable, allows modification of content. Explanation: Both StringBuilder and StringBuffer are mutable, meaning their contents can be modified after they are created.

9. API Availability

- **StringBuilder**: Similar API to StringBuffer, but no thread-safe methods.

- **StringBuffer**: Similar API to StringBuilder, with thread-safety mechanisms. Explanation: The APIs of both classes are almost identical, but StringBuffer includes synchronized methods for thread safety, while StringBuilder does not.

10. Recommended For

- **StringBuilder**: Single-threaded applications or scenarios where synchronization is not needed.
- **StringBuffer**: Multi-threaded applications where multiple threads need to access and modify the string. Explanation: StringBuilder is best used in environments where performance is critical and there is no need for thread safety, while StringBuffer is better suited for scenarios where thread safety is essential.

EXAMPLE PROGRAM

```
public class StringComparison {

public static void main(String[] args) {

long startTime = System.currentTimeMillis();

StringBuilder sb = new StringBuilder();

for (int i = 0; i < 100000; i++) {

sb.append("Hello");

}

long endTime = System.currentTimeMillis();

System.out.println("Time taken by StringBuilder (single-threaded): " + (endTime - startTime) +
"ms");

Thread thread1 = new Thread(() -> {
```

```
long threadStart = System.currentTimeMillis();

StringBuffer sbf = new StringBuffer();

for (int i = 0; i < 100000; i++) {

sbf.append("World");

}

long threadEnd = System.currentTimeMillis();

System.out.println("Time taken by StringBuffer (multi-threaded): " + (threadEnd - threadStart) +
"ms");

});

thread1.start();

try {

thread1.join();

} catch (InterruptedException e) {

e.printStackTrace();

} } }
```

OUTPUT:-

```
Time taken by StringBuilder (single-threaded): 9ms
Time taken by StringBuffer (multi-threaded): 6ms
```

MATH CLASS IN JAVA

The **Math** class in Java is part of the java.lang package, which provides a collection of static methods to perform common mathematical operations. Since the Math class consists of only static methods, you do not need to create an object of this class to use its methods.

Detailed Features:

- The **Math** class contains methods for performing basic arithmetic operations such as addition, subtraction, multiplication, division, and more.
- It also provides methods for more complex operations such as exponentiation (pow()), logarithms (log()), trigonometric functions (sin(), cos(), tan()), and rounding operations (round(), ceil(), floor()).
- The Math class also includes a **random number generator** through the Math.random() method, which returns a pseudo-random number between 0.0 and 1.0.
- **Constants**: The Math class has constants such as **Math.PI** (the value of Pi) and **Math.E** (Euler's number), which are useful in scientific calculations.

EXAMPLE PROGRAM

```
public class MathExample {

    public static void main(String[] args) {

        double powerResult = Math.pow(2, 5);

        System.out.println("2 raised to the power of 5: " + powerResult);

        double sqrtResult = Math.sqrt(25);

        System.out.println("Square root of 25: " + sqrtResult);

        double randomValue = Math.random();

        System.out.println("Random Value: " + randomValue);

        int absValue = Math.abs(-100);
        System.out.println("Absolute value of -100: " + absValue);
    }
}
```

OUTPUT:-

```
2 raised to the power of 5: 32.0
Square root of 25: 5.0
Random Value: 0.683151671726452
Absolute value of -100: 100
```

SCANNER CLASS IN JAVA

The **Scanner** class, located in the java.util package, is one of the most commonly used classes for getting input from users. It allows reading various types of data such as strings, integers, floats, booleans, etc. The Scanner class can be used to read input from different sources such as keyboard input, files, or streams.

Detailed Features:

- **Input Handling**: Scanner simplifies handling user input in various formats. It reads tokens (chunks of text), interprets them into specific data types, and allows you to process the data accordingly.
- **Reading Different Data Types**: The Scanner class offers methods such as nextInt(), nextDouble(), nextLine(), and nextBoolean() to read integers, doubles, strings, and booleans, respectively.
- **Delimiters**: You can specify delimiters using the useDelimiter() method, which allows the Scanner to split input based on specific characters or patterns.

- **Error Handling**: The Scanner class also allows you to handle invalid input and provides methods like hasNext() to check if more tokens are available.

EXAMPLE PROGRAMS

```
import java.util.Scanner;

public class ScannerExample {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

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

String name = scanner.nextLine();

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

int age = scanner.nextInt();

System.out.println("Hello, " + name + ". You are " + age + " years old.");

scanner.close();

}

}
```

OUTPUT:-

```
Enter your name: java
Enter your age: 22

Hello, java. You are 22 years old.
```

RANDOM CLASS IN JAVA

The **Random** class is part of the java.util package and is used to generate pseudo-random numbers in Java. Unlike Math.random(), which generates a random number between 0.0 and 1.0, the Random class provides more flexibility by allowing the generation of random integers, doubles, booleans, and more within a specified range.

Detailed Features:

- **Random Integer Generation**: The nextInt() method can generate random integers. You can specify a range for the random integer using nextInt(n), which generates a value between 0 (inclusive) and n (exclusive).
- **Random Float and Double**: nextFloat() and nextDouble() generate random floating-point numbers. The nextDouble() method, for example, generates a number between 0.0 (inclusive) and 1.0.
- **Random Boolean**: The nextBoolean() method generates a random boolean value (either true or false).
- **Seeded Random Numbers**: You can provide a seed value to the Random constructor for generating repeatable random sequences.

EXAMPLE PROGRAM

```
import java.util.Random;

public class RandomExample {

    public static void main(String[] args) {

        Random random = new Random();

        int randomInt = random.nextInt(100);
        System.out.println("Random Integer: " + randomInt);

        boolean randomBool = random.nextBoolean();

        System.out.println("Random Boolean: " + randomBool);

        double randomDouble = random.nextDouble();

        System.out.println("Random Double: " + randomDouble);

    }}
```

OUTPUT:-

```
Random Integer: 45
Random Boolean: false
Random Double: 0.16687296510690963
```

LOGGER CLASS IN JAVA

The **Logger** class is part of the java.util.logging package and is used for logging messages during program execution. It is a powerful tool for tracking and debugging Java applications by allowing developers to log information, warnings, errors, and other types of messages with different severity levels.

Detailed Features:

- **Logging Levels**: The Logger class supports multiple logging levels, such as INFO, WARNING, SEVERE, CONFIG, FINE, and FINER. Each level indicates the severity of the logged message.
- **Logging Handlers**: You can configure logging handlers that determine where the log messages are output (console, file, etc.).

- **Configurable Log Format**: The format of the logged messages can be customized to include information such as timestamps, log levels, and message details.
- **Exception Logging**: You can log exceptions with detailed stack traces to facilitate easier debugging.

EXAMPLE PROGRAM

```
import java.util.logging.*;

public class LoggerExample {

   private static final Logger logger = Logger.getLogger(LoggerExample.class.getName());

   public static void main(String[] args) {

      logger.setLevel(Level.ALL);

      logger.info("This is an INFO level message.");

      logger.warning("This is a WARNING level message.");

      logger.severe("This is a SEVERE level message.");

      try {

         int result = 10 / 0;

      } catch (ArithmeticException e) {

         logger.log(Level.SEVERE, "Error occurred: Division by zero", e);

      }

   }
}
```

OUTPUT:-

```
SEVERE: Error occurred: Division by zero
java.lang.ArithmeticException: / by zero
	at ebook.LoggerExample.main(LoggerExample.java:12)
```

EXCEPTION HANDLING - GRACEFUL ERROR HANDLING

Exception handling allows programs to handle errors gracefully without crashing, providing a better user experience.

EXAMPLE PROGRAM:-

```
public class BankAccount {
    private double balance;
    public BankAccount(double initialBalance) {
        if (initialBalance < 0) {
            throw new IllegalArgumentException("Initial balance cannot be negative");
        }
        this.balance = initialBalance;
    }
    public void withdraw(double amount) {
        if (amount > balance) {
            throw new InsufficientFundsException("Insufficient funds for withdrawal");
        }
        balance -= amount;
        System.out.println("Withdrawal successful. Remaining balance: $" + balance);
    }

    public double getBalance() {
        return balance;
    }
    public static void main(String[] args) {
        try {
            BankAccount account = new BankAccount(500);
            account.withdraw(100);
            account.withdraw(600);  // This will trigger the exception
        } catch (InsufficientFundsException e) {
            System.err.println(e.getMessage());
        } catch (IllegalArgumentException e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
class InsufficientFundsException extends RuntimeException {
    public InsufficientFundsException(String message) {
        super(message);
    }
}
```

VARIABLES IN JAVA

In Java, a **variable** is a container or a memory location used to store data that can be accessed and manipulated throughout the execution of a program. Each variable in Java has a name, a data type, and a value. The data type defines the kind of data a variable can hold, such as an integer, a floating-point number, or a string. The name, also known as the identifier, is used to refer to that specific piece of data within the program. Java is a **strongly-typed** language, meaning each variable must be declared with a specific type before it is used. Once declared, a variable can be initialized with a value and can be changed or updated as needed.

Variables are essential in programming as they allow you to store information that can change over time, such as the user's input, the result of a calculation, or the state of an application. In Java, variables play a critical role in controlling the flow of data within programs.

Types of Variables in Java

Java variables are categorized based on their scope, lifetime, and memory allocation. The three main types of variables in Java are:

1. **Local Variables**
2. **Instance Variables**
3. **Class (Static) Variables**

1. LOCAL VARIABLES

Local variables are variables that are declared inside methods, constructors, or blocks. They are only accessible within the method, constructor, or block where they are declared. Local variables do not have default values, so they must be explicitly initialized before use. Once the method or block finishes execution, the local variable is destroyed, making its lifetime very limited.

Local variables are typically used for temporary data storage within methods. They allow for efficient memory management because they only exist during the execution of the method or block in which they are declared.

EXAMPLE PROGRAM

```
public class LocalVariableExample {

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

```
int x = 10;

System.out.println(x);

}

}
```

OUTPUT:-

```
10
```

2. INSTANCE VARIABLES

Instance variables are variables declared inside a class but outside of any method, constructor, or block. These variables are associated with instances of the class (objects). Each object created from the class has its own copy of the instance variables, which means that different objects can hold different values for these variables.

Instance variables are initialized with default values if no explicit initialization is provided. For example, an integer instance variable will have a default value of 0. These variables are essential for storing the state of an object.

EXAMPLE PROGRAM

```
public class InstanceVariableExample {

int number=53;

public void display() {

System.out.println("The number is: " + number);
}

public static void main(String[] args) {

InstanceVariableExample obj = new InstanceVariableExample();

obj.display();

}}
```

OUTPUT:-

```
The number is: 53
```

3. CLASS (STATIC) VARIABLES

Class variables, also known as static variables, are declared with the static keyword. Unlike instance variables, class variables are shared by all objects of the class. This means that a single copy of the static variable exists, regardless of how many instances of the class are created. Static variables are often used to store properties that are common to all objects of the class.

Static variables can be accessed directly through the class name, or through any instance of the class. These variables are initialized when the class is loaded into memory, and they exist for the lifetime of the program.

EXAMPLE PROGRAM

```
public class StaticVariableExample {

    static int count = 0;

    public void increment() {

        count++;
    }

    public static void main(String[] args) {

        StaticVariableExample obj1 = new StaticVariableExample();

        StaticVariableExample obj2 = new StaticVariableExample();

        obj1.increment();

        obj2.increment();

        System.out.println("Count: " + count);

    }}
```

OUTPUT:-

```
Count: 2
```

THIS KEYWORD

In Java, the this keyword is a reference variable that refers to the current instance of the class. It is used within instance methods or constructors to refer to the current object of the class. The this keyword helps to differentiate between instance variables and local variables or parameters with the same name. It can also be used to invoke instance methods and constructors from the current object.

EXAMPLE PROGRAM

```
public class BankAccount {

private String accountHolderName;

private double balance;

public BankAccount(String accountHolderName, double initialDeposit) {

this.accountHolderName = accountHolderName;
this.balance = initialDeposit;

}

public BankAccount(String accountHolderName) {

this(accountHolderName, 0.0);

}

public BankAccount deposit(double amount) {

if (amount > 0) {

this.balance += amount;

}

return this;

}

public BankAccount withdraw(double amount) {

if (amount > 0 && amount <= this.balance) {

this.balance -= amount;

} else {

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

}

return this;

}

public void displayAccountDetails() {

System.out.println("Account Holder: " + this.accountHolderName);
System.out.println("Balance: " + this.balance);

}

public static void main(String[] args) {

BankAccount account = new BankAccount("John Doe");

account.deposit(500.0).withdraw(200.0).displayAccountDetails();

BankAccount account2 = new BankAccount("Jane Doe", 1000.0);

account2.deposit(200.0).withdraw(50.0).displayAccountDetails();

}}
```

OUTPUT:-

```
Account Holder: John Doe
Balance: 300.0
Account Holder: Jane Doe
Balance: 1150.0
```

TYPECASTING IN JAVA

Typecasting is the process of converting one data type to another in Java. It is particularly useful when dealing with different data types and ensuring that operations are performed with compatible types. Typecasting can be done either **implicitly** (automatically) or **explicitly** (manually).

Types of Typecasting in Java:

1. Implicit Typecasting (Widening)
2. Explicit Typecasting (Narrowing)

1. IMPLICIT TYPECASTING (WIDENING)

Implicit typecasting, also known as **widening**, happens automatically when a smaller data type is assigned to a larger data type. In this case, no explicit cast is needed because Java automatically converts the value to the appropriate type. This happens when there is no loss of data or precision, and the conversion is safe.

EXAMPLE PROGRAM

```
public class ImplicitCasting {

public static void main(String[] args) {

int num = 10;

double result = num;

System.out.println("The result is: " + result);

}}
```

OUTPUT:-

```
The result is: 10.0
```

2. EXPLICIT TYPECASTING (NARROWING)

Explicit typecasting, also known as **narrowing**, occurs when a larger data type is converted into a smaller data type. This requires the programmer to manually cast the data type using parentheses. Narrowing may result in a loss of data, so it needs to be done carefully. If the value being converted is too large to fit in the smaller data type, it may cause an overflow or other unexpected behavior.

EXAMPLE PROGRAM

```
public class ExplicitCasting {

public static void main(String[] args) {

double pi = 3.14159;

int intPi = (int) pi;

System.out.println("The integer value of pi is: " + intPi);

}}
```

OUTPUT:-

```
The integer value of pi is: 3
```

CONSTRUCTORS IN JAVA

A **constructor** in Java is a special method that initializes an object when it is created. Unlike normal methods, constructors have a unique purpose: they prepare an object for use by initializing its state (i.e., assigning values to its instance variables or performing setup tasks). The name of a constructor must match the name of its class, and it cannot have a return type.

Key Characteristics of Constructors:

1. **Name Matches Class:** The name of the constructor must be the same as the class name.
2. **No Return Type:** Constructors do not return any value, not even void.

3. **Automatic Invocation:** A constructor is called automatically when an object is created using the new keyword.
4. **Initialization:** Constructors are primarily used to initialize the data of an object.
5. **No Inheritance:** Constructors are not inherited, but a subclass can call a superclass's constructor using super().
6. **Overloading:** A class can have multiple constructors with different parameter lists, a concept known as constructor overloading.

Types of Constructors in Java

Java supports three main types of constructors, which are:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

DEFAULT CONSTRUCTOR

A **default constructor** in Java is a no-argument constructor that the compiler provides automatically if no other constructors are explicitly defined in a class. Its primary purpose is to initialize an object with default values and ensure that object creation is possible even when no specific initialization is provided.

Characteristics of a Default Constructor

1. **No Arguments:** The default constructor does not take any parameters.
2. **Compiler-Generated:** If no constructors are defined in a class, the Java compiler automatically creates a default constructor during the compilation process.
3. **Implicit Definition:** The programmer does not need to explicitly define it unless specific actions are needed during object initialization.

4. **Default Values:** Instance variables are initialized to their default values:
   - Numeric types (e.g., int, double) are set to 0 or 0.0.
   - Boolean is set to false.
   - Reference types (e.g., String, objects) are initialized to null.

EXAMPLE PROGRAM

```
class IndianRupeeWallet {

String ownerName;

int rupeeBalance;

public IndianRupeeWallet() {

ownerName = "Default Owner";

rupeeBalance = 500;

}

public void displayWalletDetails() {

System.out.println("Owner Name: " + ownerName);

System.out.println("Rupee Balance: ₹" + rupeeBalance);

} }

public class RupeeWalletDemo {

public static void main(String[] args) {

IndianRupeeWallet wallet1 = new IndianRupeeWallet();
wallet1.displayWalletDetails();

IndianRupeeWallet wallet2 = new IndianRupeeWallet();

wallet2.displayWalletDetails();

}}
```

OUTPUT:-

```
Owner Name: Default Owner
Rupee Balance: ₹500
Owner Name: Default Owner
Rupee Balance: ₹500
```

PARAMETERIZED CONSTRUCTOR

A **parameterized constructor** in Java is a constructor that accepts arguments to initialize an object with specific values at the time of its creation. Unlike a default constructor, which assigns default or pre-defined values, a parameterized constructor allows for more flexibility and customization.

Characteristics of Parameterized Constructor in Java

Here are the key characteristics of a parameterized constructor:

1. **Accepts Parameters:**
    - A parameterized constructor takes arguments that are used to initialize the object's fields. These parameters can be of any data type, including primitive types or objects.

2. **Facilitates Object Customization:**
   - It allows the creation of objects with specific, unique values instead of relying on default or predefined values.
3. **Avoids Redundancy:**
   - Directly assigns values to the attributes at the time of object creation, eliminating the need for setter methods.
4. **Ensures Complete Initialization:**
   - Enforces the creation of fully initialized objects by requiring all necessary values at the time of instantiation.

EXAMPLE PROGRAM

```
class LibraryBook {

String title;

String author;

int pages;

public LibraryBook(String title, String author, int pages) {

this.title = title;

this.author = author;

this.pages = pages;

}

public void displayDetails() {

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

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

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

}

}

public class LibrarySystem {

public static void main(String[] args) {

LibraryBook book1 = new LibraryBook("The Alchemist", "Paulo Coelho", 208);

LibraryBook book2 = new LibraryBook("To Kill a Mockingbird", "Harper Lee", 281);

LibraryBook book3 = new LibraryBook("1984", "George Orwell", 328);

book1.displayDetails();

book2.displayDetails();

book3.displayDetails();

}

}
```

OUTPUT:-

```
Title: The Alchemist
Author: Paulo Coelho
Pages: 208
```

```
--------------------------
Title: To Kill a Mockingbird
Author: Harper Lee
Pages: 281
--------------------------
Title: 1984
Author: George Orwell
Pages: 328
--------------------------
```

COPY CONSTRUCTOR

A **copy constructor** in Java is a special constructor used to create a new object as a copy of an existing object. It duplicates the fields of an existing object into a new object, providing a convenient way to clone or replicate objects while maintaining their independence.

Characteristics of Copy Constructor

1. **Takes an Object of the Same Class as an Argument:**
   - The parameter of a copy constructor must be an object of the same class.
2. **Customizable Behavior:**
   - Allows developers to define how fields, including references, should be copied (shallow or deep).
3. **Independent Memory Allocation:**
   - Creates a new memory block for the copied object, ensuring no unintended sharing of references.
4. **Manual Implementation:**

- Unlike some programming languages, Java does not have a default copy constructor. It must be implemented manually in the class.

5. **Supports Both Shallow and Deep Copies:**
   - By default, fields can be directly assigned (shallow copy), or custom logic can be added to copy objects or arrays deeply.
6. **Useful for Immutable Objects:**
   - Often used to replicate objects whose fields should not be modified after creation.

EXAMPLE PROGRAM

```
class BankAccount {
String accountHolder;
double balance;
public BankAccount(String accountHolder, double balance) {
this.accountHolder = accountHolder;
this.balance = balance;
}
public BankAccount(BankAccount other) {
this.accountHolder = other.accountHolder;
this.balance = other.balance;
}
public void displayDetails() {
System.out.println("Account Holder: " + accountHolder);
System.out.println("Balance: ₹" + balance);
}
public void withdraw(double amount) {
if (amount <= balance) {
balance -= amount;
System.out.println("Withdrawal of ₹" + amount + " successful.");
} else {
```

```
System.out.println("Insufficient funds!");
}}}
```

OUTPUT:-

```
Original Account Details:
Account Holder: John Doe
Balance: ₹50000.0
Copied Account Details:
Account Holder: John Doe
Balance: ₹50000.0
Withdrawal of ₹10000.0 successful.
Updated Original Account Details:
Account Holder: John Doe
Balance: ₹40000.0
Copied Account Details (Unchanged):
Account Holder: John Doe
Balance: ₹50000.0
```

METHODS IN JAVA

In Java, a **method** is a block of code designed to perform a particular task. It is one of the building blocks of Java programs, allowing for code reusability, abstraction, and organization. Methods are defined inside a class and can be called upon to perform operations on data or to carry out specific actions.

Java's object-oriented nature encourages the use of methods to define behaviors associated with objects. Methods allow programmers to write modular, reusable, and readable code. They can be invoked to carry out tasks like calculations, manipulating object states, or controlling program flow.

Key Components of a Method in Java:

1. **Method Declaration:**
   - The method declaration includes the **access modifier**, **return type**, **method name**, and **parameter list**. These are the essential components of the method.
2. **Access Modifier:**
   - The access modifier determines the visibility of the method within the class or across the program. Common access modifiers include:
     - public: The method is accessible from anywhere.
     - private: The method is accessible only within the class.
     - protected: The method is accessible within the package and by subclasses.
     - Default (no modifier): The method is accessible within the package.
3. **Return Type:**
   - The return type specifies the type of value the method will return. It can be a primitive type (like int, float, char), an object (like String, ArrayList), or void if the method doesn't return anything.
4. **Method Name:**
   - The method name should follow Java naming conventions (camelCase) and should describe the task the method performs.
5. **Parameters (Optional):**
   - Methods may accept one or more parameters that act as input to the method. These are placed inside the parentheses and allow data to be passed into the method for processing.
6. **Method Body:**
   - The method body contains the logic or code that defines the behavior of the method. It is enclosed in {}.

TYPES OF METHODS IN JAVA

✓ Instance Method

- ✓ Static Method
- ✓ Abstract Method
- ✓ Final Method

INSTANCE METHOD

In Java, an **Instance Method** is a method that belongs to an instance of a class, meaning it can only be invoked on an object of the class, not the class itself. These methods have access to the instance variables (or fields) and other instance methods within the class. The key characteristic of an instance method is that it operates on data that is specific to an object of the class, rather than the class itself.

Instance methods are commonly used to define behaviors or actions that depend on the state of an object. When a new object of a class is created, an instance method can be called to perform some operation related to that particular object's state.

When to Use Instance Methods?

Instance methods should be used in the following scenarios:

1. **When the behavior depends on the state of the object:** Since instance methods can access and modify instance variables, they are ideal when the behavior needs to depend on the specific state of the object. For example, if a method calculates the area of a circle, the radius (instance variable) of the specific circle object should be used in the calculation.
2. **When an action is tied to an individual object:** If you need to perform actions that are specific to an object rather than the class itself, an instance method is appropriate. For instance, methods that simulate actions (e.g., a car starting, a person speaking) should be instance methods, because these actions occur on individual objects.
3. **When object-oriented principles are important:** Instance methods help encapsulate functionality in the class and are aligned with key object-oriented

programming principles, such as abstraction, inheritance, and polymorphism. They allow you to structure behaviors specific to objects and help maintain the integrity of the object's state.

Syntax of Instance Method

An instance method is declared inside a class with the following syntax:

```
returnType methodName(parameters) {
   // method body
}
```

- returnType: The type of value the method returns (or void if it doesn't return anything).
- methodName: The name of the method.
- parameters: A list of parameters the method takes (if any).

EXAMPLE PROGRAM

```
class Student {

String name;

int[] scores;

public Student(String name, int[] scores) {

this.name = name;

this.scores = scores;

}

public double calculateAverage() {

int total = 0;
```

```
for (int score : scores) {
total += score;
}return total / (double) scores.length;
}public String checkPassFail() {
double average = calculateAverage();
if (average >= 50) {
return name + " has passed with an average score of " + average;
} else {
return name + " has failed with an average score of " + average;
}}
public void displayScores() {
System.out.println("Scores for " + name + ":");
for (int i = 0; i < scores.length; i++) {
System.out.println("Subject " + (i + 1) + ": " + scores[i]);
}}}public class Main {
public static void main(String[] args) {
int[] student1Scores = {85, 90, 78, 92, 88};
Student student1 = new Student("Alice", student1Scores);
```

```
int[] student2Scores = {45, 50, 40, 55, 60};

Student student2 = new Student("Bob", student2Scores);

student1.displayScores();

System.out.println(student1.checkPassFail());

student2.displayScores();

System.out.println(student2.checkPassFail());

}}
```

OUTPUT:-

```
Scores for Alice:
Subject 1: 85
Subject 2: 90
Subject 3: 78
Subject 4: 92
Subject 5: 88
Alice has passed with an average score of 86.6
Scores for Bob:
Subject 1: 45
Subject 2: 50
Subject 3: 40
Subject 4: 55
Subject 5: 60
Bob has passed with an average score of 50.0
```

STATIC METHOD

In Java, a **static method** is a method that belongs to the **class** rather than to instances of the class (objects). This means that static methods can be called directly on the class itself without the need to create an instance (object) of the class.

A static method can access only **static variables** and **static methods** within the class. It cannot access instance (non-static) variables or instance methods because instance members require an object to exist.

When to Use Static Methods:

Static methods are typically used in situations where the method functionality is **independent of instance variables**. You should use static methods when:

- The method performs a utility function that doesn’t depend on the state (or data) of any particular object.
- You want to perform operations that are the same across all instances of the class.
- You need to create helper or utility methods that can be called without creating an object of the class.

EXAMPLE PROGRAM

```
class TemperatureConverter {

public static double celsiusToFahrenheit(double celsius) {

return (celsius * 9/5) + 32;

}

public static double fahrenheitToCelsius(double fahrenheit) {

return (fahrenheit - 32) * 5/9;
```

```
}

public static double celsiusToKelvin(double celsius) {

return celsius + 273.15;

}

public static double kelvinToCelsius(double kelvin) {

return kelvin - 273.15;

}

public static double fahrenheitToKelvin(double fahrenheit) {

return (fahrenheit - 32) * 5/9 + 273.15;

}

public static double kelvinToFahrenheit(double kelvin) {

return (kelvin - 273.15) * 9/5 + 32;

}

}

public class Main {

public static void main(String[] args) {

double celsius = 25;

double fahrenheit = 77;
double kelvin = 298.15;

System.out.println(celsius + " °C is " + TemperatureConverter.celsiusToFahrenheit(celsius) + " °F");

System.out.println(fahrenheit + " °F is " + TemperatureConverter.fahrenheitToCelsius(fahrenheit) + " °C");

System.out.println(celsius + " °C is " + TemperatureConverter.celsiusToKelvin(celsius) + " K");

System.out.println(kelvin + " K is " + TemperatureConverter.kelvinToCelsius(kelvin) + " °C");

System.out.println(fahrenheit + " °F is " + TemperatureConverter.fahrenheitToKelvin(fahrenheit) + " K");

System.out.println(kelvin + " K is " + TemperatureConverter.kelvinToFahrenheit(kelvin) + " °F");

}

}
```

OUTPUT:-

```
25.0 °C is 77.0 °F
77.0 °F is 25.0 °C
25.0 °C is 298.15 K
298.15 K is 25.0 °C
77.0 °F is 298.15 K
298.15 K is 77.0 °F
```

ABSTRACT METHOD

In Java, an **abstract method** is a method that is declared without an implementation. It acts as a blueprint for subclasses, which are required to provide a specific implementation for the method. Abstract methods are used in abstract classes or interfaces. The key idea behind abstract methods is to define a **contract** that subclasses must adhere to, ensuring that certain functionality is implemented in all subclasses, while still leaving the specifics of that implementation to the subclasses.

Why Abstract Methods are Partially Abstract:

1. **Promote Code Reusability:** Abstract methods enable code reuse by allowing common functionality to be written in a base abstract class. Each subclass can implement the abstract methods to provide specific behavior. This avoids redundancy in the code. For example, different types of animals (like cats and dogs) might share the same behavior in terms of a method like makeSound(), but each will implement it differently.
2. **Enforce a Common Interface:** By using abstract methods, a parent class enforces a consistent interface (or contract) across all its subclasses. This ensures that certain operations are present across all subclasses, even if the implementations vary.
3. **Flexibility:** Abstract methods allow flexibility in design. The parent class defines what needs to be done (i.e., what the method signature is) but leaves the implementation details to be determined by each subclass. This is useful in designing flexible systems where different implementations might be needed depending on the specific subclass.
4. **Support for Polymorphism:** Abstract methods, especially when used in abstract classes or interfaces, support polymorphism. A superclass reference can point to objects of subclasses, and the method that gets called will depend on the actual object type, not the reference type. This allows for more flexible and dynamic systems.

EXAMPLE PROGRAM

```
abstract class Task {

public abstract void performTask();
```

```
}

class CodingTask extends Task {

public void performTask() {

System.out.println("Writing code for the new feature.");

}

}

class TestingTask extends Task {

public void performTask() {

System.out.println("Performing unit testing and debugging.");

}

}

class DesignTask extends Task {

public void performTask() {

System.out.println("Designing the user interface for the application.");

}

}

public class TaskManagementSystem {

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

```
Task task1 = new CodingTask();

Task task2 = new TestingTask();

Task task3 = new DesignTask();

task1.performTask();

task2.performTask();

task3.performTask();

}

}
```

OUTPUT:-

```
Writing code for the new feature.
Performing unit testing and debugging.

Designing the user interface for the application.
```

FINAL METHOD

A **final method** is a method that cannot be overridden by any subclass. This guarantees that the implementation of the method in the base class will be used without modification, no matter what class inherits from it. In other words, once a method is declared as final, it cannot be modified by subclass methods.

When to Use Final Method?

You should use a final method when:

- **You want to guarantee that a method's behavior remains unchanged across all subclasses.** For example, you may have a method that implements a fundamental algorithm, and modifying it could break important functionality.
- **You are building a secure class or library.** In cases where a method is critical for the security or integrity of the system, you would want to prevent subclasses from altering its behavior.
- **You need performance optimization.** In certain scenarios, marking methods as final helps the JVM optimize method calls, as it can safely predict that the method will not be overridden.

EXAMPLE PROGRAM

```
class PasswordManager {

private String password;

public PasswordManager(String password) {

this.password = password;

}

public final boolean validatePassword() {

if (password.length() < 8) {

System.out.println("Password must be at least 8 characters long.");

return false;

}

if (!password.matches(".*[A-Z].*")) {

System.out.println("Password must contain at least one uppercase letter.");
return false;
}
if (!password.matches(".*[0-9].*")) {
System.out.println("Password must contain at least one number.");
return false;
}
return true;
}
}
class AdminPasswordManager extends PasswordManager {
public AdminPasswordManager(String password) {
super(password);
}
}
class UserPasswordManager extends PasswordManager {
public UserPasswordManager(String password) {
super(password);
} }
public class PasswordTest {
public static void main(String[] args) {
PasswordManager admin = new AdminPasswordManager("Admin1234");
PasswordManager user = new UserPasswordManager("User1234");

System.out.println("Admin password validation: " + admin.validatePassword());

System.out.println("User password validation: " + user.validatePassword());

}}
```

OUTPUT:-

```
Admin password validation: true
User password validation: true
```

MEMORY ALLOCATION IN JAVA

Memory management in Java is crucial for efficient execution of programs. Understanding how the Java Virtual Machine (JVM) manages memory through the **Heap**, **Stack**, and **Method Area** is essential for writing optimal Java applications. Each memory area is designed for specific purposes and plays a significant role in the performance and behavior of a program.

- **Heap**
- **Stack**
- **Method Area**

HEAP AREA

The **Heap** is the region of memory used for **dynamic memory allocation**. It is where objects and arrays are stored. When you use the new keyword to create an object, the memory for that object is allocated in the heap. Similarly, arrays, whether they contain primitive data types or objects, are also stored in the heap. Unlike the stack, which is used for temporary memory, the heap is used for **long-lived objects** that might need to be accessed across different parts of the program.

- **Dynamic Allocation**: The size of the heap is not fixed. Objects can be dynamically created at runtime, and the heap grows as needed to accommodate the new objects.
- **Garbage Collection**: The heap is managed by the **Garbage Collector (GC)**. Objects that are no longer referenced by any part of the program become **garbage** and are automatically removed by the GC to free up memory.

Use Cases:

- When you create a new object or array (e.g., new String()), the memory for that object is allocated in the heap.
- Objects in the heap can be shared across methods and even different threads.

Size and Management:

- The size of the heap can be controlled via JVM options (-Xms for initial heap size and -Xmx for maximum heap size).
- The heap is **garbage-collected**, meaning unused memory is automatically reclaimed to avoid memory leaks.

STACK AREA

The **Stack** is used for **temporary memory allocation**. It stores **local variables**, **method call frames**, and **return addresses**. Every time a method is invoked, a new **stack frame** is created to hold the method's local variables and execution details. When the method execution completes, its stack frame is popped off the stack, freeing the memory.

- **Last-In, First-Out (LIFO) Structure**: The stack follows the LIFO order. When a method is called, its stack frame is pushed onto the top of the stack, and once the method returns, its frame is popped from the stack.
- **Automatic Deallocation**: Variables stored in the stack are automatically deallocated when the method completes.

Use Cases:

- The stack stores data that is only needed for a specific method execution, such as local variables and parameters.
- Stack memory is used for **recursion** and method calls.

Size and Management:

- The stack is smaller than the heap and is managed automatically.
- Stack space is limited, and exceeding this space (e.g., by deep recursion) results in a **StackOverflowError**.

<u>METHOD AREA</u>

The **Method Area** (or **Metaspace** in modern versions of Java) is a special region of memory where **class-level data** is stored. This includes the bytecode of classes, method definitions, static variables, and constant pools. Unlike heap and stack memory, the method area is **shared among all threads**.

- **Class-Level Data**: It holds information about loaded classes, including class metadata, method and field data, and constant values.
- **Runtime Constant Pool**: This area also stores constants and references, such as string literals and method references, used by the class during runtime.

Use Cases:

- The method area is crucial for the **JVM class loading process**. Whenever a class is loaded into memory, its metadata and bytecode are stored in the method area.
- It contains **static variables** shared by all instances of a class and **compiled bytecode** for methods.

Size and Management:

- The size of the method area is dynamic in modern Java, and in the case of Metaspace, it grows automatically.
- In earlier versions of Java, the method area had a fixed size (managed by the -XX:MaxPermSize option).

CONDITIONAL STATEMENT IN JAVA

Conditional statements in Java are used to execute specific blocks of code based on a given condition or expression's evaluation. They allow the program to make decisions and perform different actions depending on whether certain conditions are met. These statements are fundamental for controlling the flow of execution and are part of decision-making logic in programming.

There are several types of conditional statements in Java, each serving different purposes depending on the complexity and requirements of the program.

The most common conditional constructs include

- If
- If…Else
- Nested If
- Switch

IF

The if statement in Java is a decision-making construct used to execute a particular block of code only when a specific condition is satisfied. It evaluates a condition, and if that condition is true, the block of code within the if statement is executed. If the condition is false, the program simply bypasses the block and continues with the rest of the code. This provides a means for a program to make decisions based on dynamic inputs or states.

SYNTAX:

```
if (condition) {
    // Code to execute if condition is true
}
```

EXAMPLE PROGRAM

```
public class LeapYearChecker {
    public static void main(String[] args) {
        int year = 2025;
        if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {
            System.out.println(year + " is a Leap Year.");
        } else {
            System.out.println(year + " is not a Leap Year.");
        }
    }
}
```

OUTPUT:-

```
2025 is not a Leap Year.
```

IF…Else

In Java, the if-else statement is a control flow statement used to execute one block of code among two alternatives, depending on whether a specified condition evaluates to true or false. It allows the program to make decisions, providing a way to execute different pieces of code based on different conditions.

The basic idea behind if-else is simple: if a condition evaluates to true, execute the code inside the if block; otherwise, execute the code inside the else block. This makes it a fundamental control structure for branching logic in Java programs.

SYNTAX

```
if (condition) {
    // Code block if the condition is true
} else {
    // Code block if the condition is false
}
```

EXAMPLE PROGRAM

```
public class VowelConsonantChecker {
    public static void main(String[] args) {
        char letter = 'B';
        if (letter == 'A' || letter == 'E' || letter == 'I' || letter == 'O' || letter == 'U' ||
            letter == 'a' || letter == 'e' || letter == 'i' || letter == 'o' || letter == 'u') {
            System.out.println(letter + " is a Vowel.");
        } else {
            System.out.println(letter + " is a Consonant.");
        }
    }
}
```

OUTPUT:-

```
B is a Consonant.
```

NESTED IF

A **nested if** statement in Java is a logical control structure where one if statement is placed inside another. This structure is particularly useful for handling decision-making scenarios that depend on multiple conditions evaluated sequentially. Each inner if block is processed only when the outer if condition is true, making nested if statements a powerful tool for implementing hierarchical or tiered logic.

SYNTAX

```
if (condition1) {
    // Outer if block
    if (condition2) {
        // Inner if block
        // Code to execute if both condition1 and condition2 are true
    } else {
        // Inner else block (optional)
        // Code to execute if condition2 is false
    }
} else {
    // Outer else block (optional)
    // Code to execute if condition1 is false
}
```

EXAMPLE PROGRAM

```
public class AdventureSuitabilityChecker {
    public static void main(String[] args) {
        int age = 25;
        boolean goodWeather = true;
        boolean isFit = true;
        if (age >= 18) {
            if (goodWeather) {
                if (isFit) {
                    System.out.println("You are suitable for the adventure trip!");
                } else {
                    System.out.println("You need to improve your fitness level for this trip.");
                }
            } else {
                System.out.println("The weather is not favorable for the trip.");
            }
        } else {
            System.out.println("You are too young for the adventure trip.");
        }   }}
```

OUTPUT:-

```
You are suitable for the adventure trip!
```

SWITCH

The **switch statement** in Java is a versatile decision-making control structure that simplifies handling multiple conditions. It evaluates a single expression and matches its result against a list of predefined values called **cases**. When a match is found, the corresponding block of code executes. This structure is a clean, concise alternative to multiple if-else conditions, particularly when testing a variable for equality against several possible values.

SYNTAX:-

```
switch (expression) {
    case value1:
        // Code block for case value1
        break;
    case value2:
        // Code block for case value2
        break;
    // More cases as needed
    default:
        // Code block for unmatched cases
}
```

EXAMPLE PROGRAM:-

```
public class DayTypeChecker {
    public static void main(String[] args) {
        int dayNumber = 7;
        String dayType;
        String dayName;
        switch (dayNumber) {
            case 1:
                dayName = "Monday";
                dayType = "Weekday";
                break;
            case 2:
```

```
    dayName = "Tuesday";

    dayType = "Weekday";

    break;

case 3:

    dayName = "Wednesday";

    dayType = "Weekday";

    break;

case 4:

    dayName = "Thursday";

    dayType = "Weekday";

    break;

case 5:

    dayName = "Friday";

    dayType = "Weekday";

    break;

case 6:

    dayName = "Saturday";

    dayType = "Weekend";

    break;

case 7:

    dayName = "Sunday";
```

```
            dayType = "Weekend";
            break;
        default:
            dayName = "Invalid Day";
            dayType = "Unknown";
    }
    System.out.println("Day: " + dayName);
    System.out.println("Type: " + dayType);
}}
```

<u>OUTPUT:-</u>

```
Day: Sunday
Type: Weekend
```

<u>LOOPS IN JAVA</u>

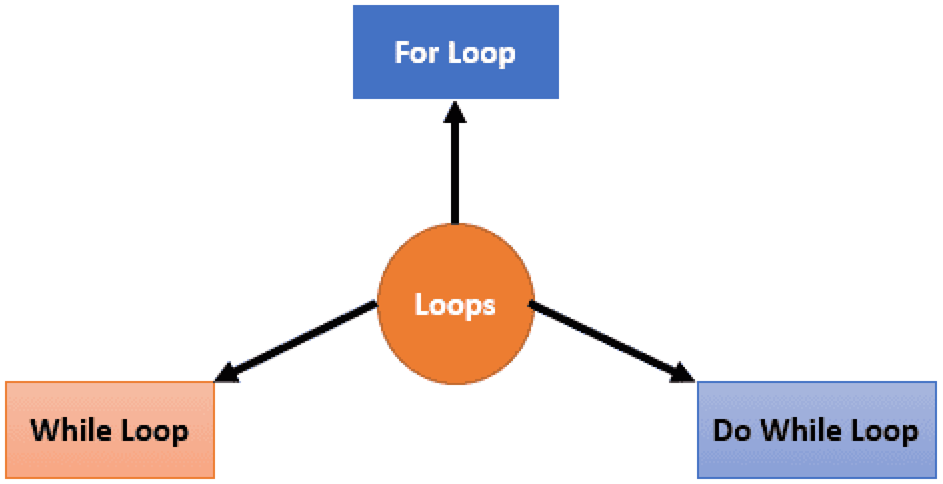
In Java, loops are fundamental constructs that enable a program to execute a block of code multiple times, based on specific conditions. This ability to repeat tasks reduces code duplication and promotes efficiency by eliminating the need for manually writing repetitive code. Loops are essential when working with large datasets or performing tasks that require repetitive actions, such as processing user input or iterating through elements in a collection. By allowing code to run continuously or a fixed number of times, loops make Java programs more concise, readable, and maintainable.

Why Loops are Important in Java:

- **Automation of Repetitive Tasks**: Loops enable the automation of tasks that require repeated actions. This is especially useful in scenarios like processing user input, calculating results for multiple entries, or traversing data structures.
- **Optimization of Code**: Without loops, code would become bloated with repetitive lines. By condensing repetitive logic into a single loop, Java reduces redundancy and helps keep the codebase clean and maintainable.
- **Handling Dynamic Data**: Loops make it easy to handle situations where the exact number of iterations is not known. They offer flexibility, allowing the program to keep running until specific conditions are met (such as fetching data from a server or processing all elements in an array).
- **Efficient Use of Resources**: When a loop runs, only a minimal amount of resources is needed for each iteration. For example, working with large datasets is more efficient when each element is processed individually in a loop, reducing memory overhead.

TYPES OF LOOPS IN JAVA

- For Loop
- While loop
- Do…while loop



For Loop

The **for loop** in Java is one of the most commonly used loops because of its clear and compact structure, making it ideal for scenarios where the number of iterations is known in advance. It's a control flow statement used to repeat a block of code multiple times. The loop works by executing the block of code as long as the specified condition evaluates to true. The loop automatically handles the initialization, condition checking, and update, reducing the complexity of the code.

SYNTAX

```
for (initialization; condition; update) {
   // code block to be executed
}
```

Initialization:

- ✓ This part is executed **only once**, when the loop starts.

- ✓ It is typically used to declare and initialize the loop variable (also known as the counter).
- ✓ For example, int i = 0; initializes the loop counter to zero.

Condition:

- ✓ The loop will **continue to run** as long as the condition evaluates to true.
- ✓ After each iteration, the condition is checked before executing the loop body again. If the condition becomes false, the loop terminates.
- ✓ For example, i < 10; ensures that the loop runs as long as i is less than 10.

Update:

- ✓ After each iteration, this part is executed.
- ✓ Typically, it modifies the loop counter (increasing or decreasing the value of the variable).
- ✓ For example, i++ increments the counter by 1 after each loop iteration.

EXAMPLE PROGRAM

```
public class FibonacciSeries {
    public static void main(String[] args) {
        int n = 10;
        int first = 0, second = 1;
        System.out.println("Fibonacci Series up to " + n + " terms:");
        for (int i = 1; i <= n; i++) {
            System.out.print(first + " ");
            int next = first + second;
            first = second;
            second = next;

        }

    }}
```

OUTPUT:-

```
Fibonacci Series up to 10 terms:
0 1 1 2 3 5 8 13 21 34
```

WHILE LOOP

A **while loop** allows repeated execution of a statement or block of code based on a boolean condition. The key feature of the while loop is that it evaluates the condition before entering the loop body. If the condition is true, the loop body is executed; if the condition evaluates to false, the loop terminates, and the program continues with the next statement after the loop.

This loop is ideal when you need to perform an action repeatedly but don't know how many times the action should be performed ahead of time. It keeps running until the condition specified within the loop becomes false.

SYNTAX

```
while (condition) {

    // Code to be executed

}
```

- ✓ **condition**: The condition is evaluated before each iteration. If it returns true, the loop will continue. If it evaluates to false, the loop terminates.
- ✓ **Code Block**: This is the block of code that will be executed repeatedly as long as the condition is true.

EXAMPLE PROGRAM

```
import java.util.Scanner;

public class FactorialCalculator {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        int number = 0;

        while (true) {

            System.out.print("Enter a positive integer to calculate its factorial: ");

            if (scanner.hasNextInt()) {

                number = scanner.nextInt();

                if (number < 0) {

                    System.out.println("Please enter a positive integer.");

                } else {

                    break;                }

            } else {

                System.out.println("Invalid input! Please enter a valid integer.");

                scanner.next();
            }       }
        long factorial = 1;
        int i = 1;
        while (i <= number) {
            factorial *= i;
            i++;       }
        System.out.println("The factorial of " + number + " is: " + factorial);
    }}
```

OUTPUT:-

```
Enter a positive integer to calculate its factorial: 8
The factorial of 8 is: 40320
```

DO…WHILE LOOP

The do-while loop in Java is a variant of the while loop. It is used when you want to execute a block of code at least once, regardless of the condition being true or false. Unlike the while loop, where the condition is checked before entering the loop, the do-while loop checks the condition **after** executing the block of code. This guarantees that the code inside the loop is executed at least once.

SYNTAX:-

```
do {
    // Code block to be executed
} while (condition);
```

do:

- ✓ This keyword marks the start of the loop block.
- ✓ The code inside the do block is executed **at least once**, regardless of whether the condition is true or false.
- ✓ The do block is where you write the logic or operations that need to be performed repeatedly.

{ ... } (Curly Braces):

- ✓ The curly braces {} define the body of the do loop. All the statements inside the braces will be executed in each iteration.
- ✓ These statements can be a single statement or multiple statements that you want to repeat.

while:

- ✓ After the do block, the while keyword is used to specify the condition that determines if the loop will continue executing.
- ✓ The condition inside the parentheses is evaluated after executing the code inside the do block.

(condition):

- ✓ The condition is a Boolean expression (i.e., an expression that evaluates to true or false).
- ✓ If the condition is true, the loop will execute again (i.e., the code inside the do block will be executed again).
- ✓ If the condition is false, the loop will terminate, and the program control will move to the next statement following the do-while loop.

Semicolon (;):

- ✓ The while (condition) part ends with a semicolon ;. This semicolon signifies the end of the loop condition statement.

EXAMPLE PROGRAM

```
public class Example {
    public static void main(String[] args) {
        int counter = 1;
        do {
            System.out.println("This is loop iteration: " + counter);
            counter++;
        } while (counter <= 5);
    }}
```

OUTPUT:-

```
This is loop iteration: 1
This is loop iteration: 2
This is loop iteration: 3
This is loop iteration: 4
This is loop iteration: 5
```

JUMP STATEMENTS

Jump statements in Java are powerful tools that control the flow of execution in your programs. They allow you to break out of loops, skip certain parts of the code, or exit from methods when certain conditions are met. This control flow is essential for creating efficient, clean, and readable code, especially when handling complex logic or managing repetitive tasks within loops.

There are two primary types of jump statements in Java:

1. break statement
2. continue statement

1.BREAK STATEMENT

The break statement in Java is a control flow mechanism that allows you to **exit a loop or switch statement prematurely**, which can significantly optimize the flow of execution in certain scenarios. By using break, you can stop further iterations of a loop or prevent code from falling through multiple switch cases, saving unnecessary computations or ensuring that only the relevant case in a switch statement is executed.

How the break Statement Works

1. In Loops

The break statement can be used inside any type of loop, including for, while, and do-while loops. When the break is executed, the loop is immediately terminated, and the control flow is transferred to the first statement **after** the loop. This is useful when the loop needs to stop under certain conditions, even if the loop's termination condition hasn't been met yet.

For example, imagine you have a loop that is designed to search through a list for a specific item. As soon as the item is found, there's no reason to continue searching, and the break statement allows you to exit the loop immediately.

2. In Switch Statements

In a switch statement, the break statement serves to **terminate the case block** and exit the switch structure. Without the break, Java will execute the code for the current case and "fall through" to execute the code of the next case, even if the case conditions do not match. The break statement ensures that once the code for a specific case has been executed, no further cases will be checked.

When to Use the break Statement

- **Exiting Loops Early**: When a condition inside a loop has been satisfied, and there's no need to continue iterating. This is commonly used in search or validation operations.
- **Preventing Fall-through in Switch Statements**: In switch blocks, break prevents the default fall-through behavior, ensuring that only one case block is executed.

<u>EXAMPLE PROGRAM</u>

```
public class BreakExample {
    public static void main(String[] args) {
        int[] numbers = {5, 10, 15, 20, 25};
        for (int i = 0; i < numbers.length; i++) {
            if (numbers[i] == 20) {
                System.out.println("Found 20! Breaking the loop...");
                break;  // Exit the loop immediately
            }
            System.out.println("Checking number: " + numbers[i]);
        }    }}
```

OUTPUT:-

```
Checking number: 5
Checking number: 10
Checking number: 15
Found 20! Breaking the loop...
```

2.CONTINUE STATEMENT

The continue statement in Java is used to **skip the current iteration** of a loop and proceed with the next iteration of the loop. When the continue statement is encountered, the remaining part of the loop body is skipped for the current iteration, and the loop moves to the next cycle. This statement is useful when you want to skip over certain conditions and continue with the loop's logic.

How the continue Statement Works

The continue statement can be used in **for, while, and do-while loops**. Depending on where it is used in the loop, it causes the loop to **skip the current iteration** and move to the next cycle.

In a for Loop:

When used in a for loop, the continue statement **skips the current iteration** and proceeds to the **increment** part of the loop before checking the loop's condition again. Essentially, it skips over the remaining statements in the loop body for that particular iteration.

In a while or do-while Loop:

When used in a while or do-while loop, the continue statement causes the loop to immediately **evaluate the condition** again, skipping the rest of the loop body for that iteration.

When to Use the continue Statement

The continue statement is useful in scenarios where:

- You want to **skip certain conditions** but still continue looping through the rest of the iterations.
- It helps reduce the need for deep nested condition checks inside loops and can make the code cleaner and more efficient by avoiding unnecessary operations.

EXAMPLE PROGRAM

```
public class ContinueExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i % 2 == 0) {
                continue;
            }
            System.out.println(i);
        }
    }
}
```

OUTPUT:-

```
1
3
5
7
```

ARRAYS IN JAVA

An **array** in Java is essentially a **container** or **data structure** that holds multiple elements of the **same data type** under a single variable name. These elements are stored in **consecutive memory locations**, and each element can be accessed via its **index**. Arrays in Java can hold either primitive types like integers, floats, and booleans, or reference types like objects and strings.

Arrays are **fixed in size**, which means once you declare and instantiate an array, its size cannot be changed during runtime. You define the number of elements it can store at the time of creation. Arrays are ideal for cases where you need to work with **a known and fixed number of items**.

Why Use Arrays in Java?

1. **Efficient Storage of Multiple Elements:** Arrays provide a means to store multiple values under one identifier. Instead of declaring several variables for each individual value, you can declare a single array and reference the elements using their index. This reduces the need for repetitive variable declarations.
2. **Faster Access to Data:** One of the main advantages of using arrays is that elements are stored in consecutive memory locations. Therefore, you can access an element in **constant time (O(1))** by providing its index. This provides a significant speed advantage, especially when dealing with large datasets where quick access to individual elements is required.
3. **Simplified Data Management:** Arrays allow you to work with a group of similar data points using loops, enabling simpler code for operations like iterating, searching, and

sorting. Instead of handling individual variables, you handle a single array, making the code more readable and manageable.

4. **Predictable Memory Usage:** Arrays provide a fixed, known memory allocation at the time of creation. This ensures that you can allocate memory efficiently without worrying about unpredictable changes in memory size, which can occur with more dynamic data structures like linked lists or hash maps.

Disadvantages of Arrays with Primitive Data Types

While arrays offer several advantages, they also come with **limitations**, especially when dealing with **primitive data types** like int, char, boolean, etc. These limitations can make arrays less suitable in some scenarios:

1. **Fixed Size**:
   - One of the most significant drawbacks of arrays in Java is their **fixed size**. Once an array is created with a specified size, it cannot be resized. If the size of the collection changes dynamically, you would need to create a new array, copy data over, and adjust its size. This rigidity makes arrays less flexible for certain applications, like handling unknown amounts of data.
2. **No Built-in Dynamic Resizing**:
   - Arrays don't provide any built-in mechanisms to **resize dynamically**. Unlike ArrayList or other collection classes in Java, which can dynamically resize when elements are added or removed, arrays require manual resizing when the data exceeds the initial allocated size. This makes arrays unsuitable for situations where the collection size changes frequently.
3. **Primitive Types Cannot Hold Null Values**:
   - Arrays that hold primitive data types (e.g., int[], char[], etc.) cannot store null values. This is a significant limitation because if you need to represent a missing or undefined value, you have to use a **default value** (such as 0 for int or false

for boolean). This can lead to confusion and possible errors when the default value is used for valid elements in the array.

4. **Memory Inefficiency with Large Arrays**:
    - Arrays of primitive types can be **memory-inefficient** when the array size is set too large but the number of valid entries is small. For instance, if you allocate an array of 1000 integers, but only use 100, you are wasting memory. Dynamic data structures like ArrayList or LinkedList are more memory-efficient because they grow and shrink as needed.
5. **No Built-in Methods for Common Operations**:
    - Arrays lack the built-in methods that make it easier to perform common operations, such as adding, removing, or resizing elements. While arrays allow direct access to their elements using indices, you must implement your own logic for tasks such as element addition, deletion, and searching, which can be cumbersome and error-prone.
6. **Multi-Dimensional Array Complexity**:
    - While Java supports **multi-dimensional arrays** (like 2D, 3D arrays), managing them can become more complex as you deal with data structures that require multiple levels of indexing. For example, in a 2D array (array of arrays), you must manage both the rows and columns separately, which can lead to confusion and a more complicated code structure.
7. **No Type Safety for Primitive Arrays**:
    - Arrays of primitive types do not offer type safety when used to hold mixed data types. You cannot store both integers and strings in the same array without converting them into reference types, which can complicate the code. If you want to store a collection of different types, you would need to use objects like Object[], but that sacrifices type safety, making it harder to detect errors at compile-time.

1. Single Dimensional Array (1D Array)

A **single-dimensional array (1D array)** is the simplest and most commonly used form of an array in Java. It is a linear collection of elements where each element is of the same data type. These elements are stored in a contiguous block of memory, which allows for efficient access based on the index of the element. In this structure, each element in the array is referenced by a single index, and it's ideal for cases where you have a collection of similar data.

Key Characteristics of Single Dimensional Arrays:

1. **Linear Storage**:
   - The elements in a 1D array are arranged in a straight line (linear sequence). This means that the elements are stored sequentially in memory, making it easy to access each element using an index.
2. **Fixed Size**:
   - When you create a 1D array in Java, you must define its size upfront. Once the size is set, it cannot be changed. This is because arrays in Java are fixed in size, and their memory allocation is contiguous.
3. **Zero-Based Indexing**:
   - In Java, arrays are indexed starting at 0. So, the first element of the array is accessed using index 0, the second element with index 1, and so on. This makes the last element of a 1D array accessible via array.length - 1.
4. **Type Consistency**:
   - All elements in a single-dimensional array must be of the same data type. This ensures that memory allocation is contiguous and efficient.
5. **Direct Memory Access**:
   - Each element in the array is stored at a specific memory location, allowing for direct access using its index. This gives arrays an edge in terms of performance, particularly when accessing or modifying elements.

When to Use a 1D Array:

A 1D array is used in various scenarios, especially when you need to represent a collection of data that shares the same type. Here are some common use cases:

- **Storing a List of Items**: If you have a list of values that belong to the same category, such as the ages of a group of people, the scores of students in an exam, or even a collection of product prices, a 1D array is the most straightforward approach.
- **Data Representation**: If you're dealing with a simple dataset where each element is distinct but of the same type, such as a set of unique IDs, names, or sensor readings, a 1D array provides a quick way to store and retrieve those elements.
- **Efficient Iteration**: For operations like summing, finding the maximum or minimum, or performing other aggregate functions, iterating over the elements of a 1D array is very efficient due to its linear structure.
- **Quick Access**: When you need fast access to specific elements by their position, such as looking up an element by index, a 1D array gives constant-time access (O(1) complexity).

Memory Representation and Access:

One of the reasons why 1D arrays are so efficient is the way they are stored in memory. When you declare an array, Java allocates a contiguous block of memory for all the elements. This is different from other data structures (like linked lists), where elements may be scattered across memory.

Each array element can be accessed directly using the index. If the starting address of the array is known, the address of any element can be calculated by adding the index multiplied by the size of the element to the base address. This is why array indexing is so fast—**it requires no traversal**. It's simply a matter of accessing the memory at a known offset.

Array Size and Limitations:

- Once you declare an array, the size is fixed. If you need a collection that can grow or shrink dynamically, you might want to use collections like ArrayList, which provides more flexibility but comes with a performance tradeoff.
- In cases where you don't know the number of elements in advance, but still need to store them, you may need to allocate a larger array and fill it dynamically or use another data structure (like ArrayList).

SYNTAX:-

1. Declaration:

dataType[] arrayName;

Explanation:

This step defines an array variable named arrayName of the specified dataType. The array is only declared here and does not hold any values until memory is allocated. The dataType can be any primitive type (e.g., int, float, char) or a reference type (e.g., String, Object).

2. Instantiation:

arrayName = new dataType[size];

Explanation:

In this step, memory is allocated to the array. The new keyword is used to create an array object in memory, and the size specifies how many elements the array will hold. The size is fixed, and once declared, it cannot be changed.

3. Combined Declaration and Instantiation:

dataType[] arrayName = new dataType[size];

Explanation:

This combines both the declaration and instantiation in a single line. It declares an array arrayName of type dataType, allocates memory for it, and defines its size in one step. This is commonly used when you know the size of the array ahead of time and want to initialize it in a single line.

EXAMPLE PROGRAM

```
public class AverageEvenNumbers {

    public static void main(String[] args) {

        int[] numbers = {12, 7, 8, 15, 22, 35, 40};

        int sum = 0;

        int count = 0;

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

            if (numbers[i] % 2 == 0) {

                sum += numbers[i];

                count++;

            }

        }

        if (count > 0) {

            double average = (double) sum / count;

            System.out.println("Average of even numbers: " + average);

        } else {
```

```
        System.out.println("No even numbers found in the array.");

    }   }}
```

OUTPUT:-

```
Average of even numbers: 20.5
```

2-D ARRAY IN JAVA

A 2D array in Java is an array whose elements are arrays themselves. This means that each element in the 2D array is a reference to another array, which can be accessed via two indices: one for the row and one for the column. The structure is similar to a matrix, where each row is an array, and the columns are the individual elements within that row.

Key Characteristics of 2D Arrays in Java

1. **Fixed Size**:
   - Once a 2D array is created, its size cannot be changed. The number of rows and columns must be specified during initialization and cannot be modified later.
2. **Contiguous Memory Allocation**:
   - A 2D array is stored in a contiguous block of memory. The array consists of an array of arrays, and each inner array represents a row. This layout allows for efficient access to elements based on their row and column indices.
3. **Indexed Access**:
   - Elements in a 2D array are accessed by two indices: one for the row and one for the column. This indexing provides direct access to any element in constant time.
4. **Row-Column Structure**:

   - The array is organized into rows and columns. It is essentially a collection of data items arranged in a grid format. This makes it suitable for representing data in a tabular or matrix-like structure.

5. **Default Values**:
   - If a 2D array is declared but not explicitly initialized, the elements will be automatically assigned default values based on the data type. For numeric types, the default value is 0, and for reference types like objects or strings, the default value is null.
6. **Memory Efficiency**:
   - 2D arrays are memory efficient when the size is known beforehand and does not change, as the elements are stored in a contiguous memory block. However, the size is static, and there is no flexibility to add or remove elements dynamically.
7. **Multi-Dimensional Arrays**:
   - Java supports arrays with more than two dimensions, referred to as multi-dimensional arrays. These arrays extend the concept of 2D arrays to higher dimensions, allowing the representation of more complex data structures.

When to Use a 2D Array

1. **Tabular Data**:
   - 2D arrays are best suited for storing tabular data where the information is organized into rows and columns. This is useful when dealing with data that follows a grid-like structure, such as schedules, records, or spreadsheets.
2. **Matrix Operations**:
   - 2D arrays are commonly used for matrix representation, where mathematical operations such as addition, subtraction, multiplication, and finding the inverse can be easily performed on them.
3. **Grid-based Systems**:

- When you need to represent a grid or a board in applications like games (e.g., chess, tic-tac-toe) or simulations (e.g., terrain mapping, maze generation), 2D arrays are an ideal choice.

4. **Image Representation**:
   - 2D arrays are frequently used to represent images, where each pixel or cell in the array corresponds to an element of the image, allowing easy manipulation of pixel data for operations like image processing or transformations.
5. **Geographical Data**:
   - 2D arrays are effective for storing geographical data such as maps, where each element in the array represents a geographic coordinate or an attribute of a specific location.
6. **Game Development**:
   - For board games or grid-based games, such as Sudoku or Minesweeper, 2D arrays provide a straightforward way to store and manage game state in a grid structure.
7. **Sorting and Searching in a Grid**:
   - 2D arrays can also be used for sorting and searching operations that involve data organized in a grid or matrix format, where algorithms like row or column-based sorting or searching are applied.

SYNTAX:-

Declaration:

```
dataType[][] arrayName;
```

- **dataType**: Specifies the type of data the array will hold, such as int, double, String, or even custom objects.
- **arrayName**: The name you assign to the array variable, which will be used to reference the array.

The declaration alone only defines the array's type and name but does not allocate memory for the array's elements.

Instantiation: After declaring an array, the next step is to allocate memory for the array. This is done by specifying the number of rows and columns the array will have.

```
arrayName = new dataType[rows][columns];
```

- **arrayName**: The previously declared array variable.
- **new**: This keyword is used to allocate memory for the array.
- **rows**: Represents the number of horizontal "sections" or "rows" in the 2D array (i.e., the first dimension).
- **columns**: Represents the number of vertical "sections" or "columns" in the 2D array (i.e., the second dimension).

This creates an array with the specified dimensions in memory, where each element is initialized with a default value (e.g., 0 for numeric types or null for reference types).

Combined Declaration and Instantiation: In Java, you can declare and instantiate a 2D array in one step by combining both processes:

```
dataType[][] arrayName = new dataType[rows][columns];
```

- **dataType[][]**: Declares the type of array, indicating that it's a 2D array.
- **arrayName**: The name assigned to the array.
- **new dataType[rows][columns]**: Allocates memory for the array with the specified number of rows and columns.

<u>EXAMPLE PROGRAM</u>

```
public class MultiplicationTable {
 public static void main(String[] args) {

     int number = 5;

     int[][] table = new int[10][2];

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

         table[i][0] = i + 1;

         table[i][1] = table[i][0] * number;

     }

     System.out.println("Multiplication Table for " + number + ":");

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

         System.out.println(table[i][0] + " x " + number + " = " + table[i][1]);

     }    }}
```

OUTPUT:-

```
Multiplication Table for 5:
1 x 5 = 5
2 x 5 = 10
3 x 5 = 15
4 x 5 = 20
5 x 5 = 25
6 x 5 = 30
7 x 5 = 35
8 x 5 = 40
9 x 5 = 45
```

10 x 5 = 50

OBJECT ORIENTED PROGRAMMING LANGUAGE

Object-Oriented Programming (OOPs) is a structured programming paradigm centered on the concept of **objects**, which encapsulate data and behavior in a single entity. Unlike procedural programming, which focuses on functions and logic flow, OOPs mimics real-world scenarios by representing entities as objects and their interactions through methods. This paradigm simplifies complex software systems by promoting modularity, reusability, and scalability.

The foundation of OOPs lies in **classes** (blueprints that define objects) and **objects** (instances of classes). Objects store attributes (data) and expose behaviors (methods), enabling developers to create intuitive models of real-world systems. This approach improves the clarity of code by organizing it into self-contained units.

Why OOPs?

1. **Modularity**: OOPs breaks down large systems into smaller, manageable modules.
2. **Reusability**: Code can be reused across different applications, reducing redundancy and effort.
3. **Scalability**: Adding new functionality is simpler due to the organized structure.
4. **Maintainability**: Changes or updates can be made with minimal impact on the entire system.

ENCAPSULATION - PROTECTING DATA

Encapsulation is about hiding the internal data and allowing controlled access via public methods.

EXAMPLE PROGRAM:-

```
public class Product {
    private String productName;
    private double price;
    private int stockQuantity;
    public Product(String productName, double price, int stockQuantity) {
        this.productName = productName;
        this.price = price;
        this.stockQuantity = stockQuantity;
    }
    public String getProductName() {
        return productName;
    }
    public void setProductName(String productName) {
        this.productName = productName;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        if (price > 0) {
            this.price = price;
        } else {
            throw new IllegalArgumentException("Price must be positive");
        }
    }
    public int getStockQuantity() {
        return stockQuantity;
    }
    public void setStockQuantity(int stockQuantity) {
        if (stockQuantity >= 0) {
            this.stockQuantity = stockQuantity;
        } else {
            throw new IllegalArgumentException("Stock quantity cannot be negative");
        }
    }
    public void displayProductInfo() {
        System.out.println("Product: " + productName + " | Price: $" + price + " | In Stock: " +
stockQuantity);
    }
}
```

INHERITANCE - REUSING CODE

Inheritance allows one class to inherit the properties and behaviors of another.

EXAMPLE PROGRAM:-

```
public class Employee {
    private String name;
    private String employeeId;

    public Employee(String name, String employeeId) {
        this.name = name;
        this.employeeId = employeeId;
    }
    public String getName() {
        return name;
    }
    public String getEmployeeId() {
        return employeeId;
    }
    public void work() {
        System.out.println(name + " is working.");
    }

    public void displayEmployeeInfo() {
        System.out.println("Employee ID: " + employeeId + ", Name: " + name);
    }
}
public class Manager extends Employee {
    private String department;

    public Manager(String name, String employeeId, String department) {
        super(name, employeeId);
        this.department = department;
    }
    @Override
    public void work() {
        super.work();
        System.out.println("Managing the " + department + " department.");
    }
    public void displayManagerInfo() {
        super.displayEmployeeInfo();
        System.out.println("Department: " + department);
    }
}
public class Main {
    public static void main(String[] args) {
        Manager manager = new Manager("Alice", "M123", "HR");
        manager.displayManagerInfo();
        manager.work();
    }}
```

POLYMORPHISM - FLEXIBILITY IN CODE

Polymorphism allows different classes to be treated as instances of the same class through inheritance, with different behaviors.

EXAMPLE PROGRAM:-

```
public class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The dog barks.");
    }
}
public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The cat meows.");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Animal dog = new Dog();
        Animal cat = new Cat();
        animal.makeSound();
        dog.makeSound();
        cat.makeSound();
    }
}
```

ABSTRACTION - HIDING IMPLEMENTATION DETAILS

Abstraction hides the complex reality while exposing only the essential parts of an object. This is achieved using abstract classes or interfaces.

EXAMPLE PROGRAM:-

```
public abstract class Vehicle {
    private String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }
    public abstract void startEngine();

    public void displayBrand() {
        System.out.println("Brand: " + brand);
    }
}
public class Car extends Vehicle {
    public Car(String brand) {
        super(brand);
    }
    @Override
    public void startEngine() {
        System.out.println("Car engine started.");
    }
}
public class Main {
    public static void main(String[] args) {
        Vehicle car = new Car("Toyota");
        car.displayBrand();
        car.startEngine();
    }
}
```

CONSTRUCTOR OVERLOADING - FLEXIBILITY IN OBJECT INITIALIZATION

Constructor overloading is a technique to provide multiple ways of creating an object, each with different parameters.

EXAMPLE PROGRAM:-

```
public class Book {
    private String title;
    private String author;
    private double price;
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
        this.price = 0.0;
    }
    public Book(String title, String author, double price) {
        this.title = title;
        this.author = author;
        this.price = price;
    }
    public void displayInfo() {
        System.out.println("Book Title: " + title + ", Author: " + author + ", Price: $" + price);
    }
}
public class Main {
    public static void main(String[] args) {
        Book book1 = new Book("Effective Java", "Joshua Bloch");
        Book book2 = new Book("Clean Code", "Robert C. Martin", 39.99);
        book1.displayInfo();
        book2.displayInfo();    } }
```

THIS KEYWORD - REFERRING TO CURRENT OBJECT

The this keyword refers to the current instance of the class, which is useful when differentiating between instance variables and method parameters.

EXAMPLE PROGRAM:-

```
public class Rectangle {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    public double calculateArea() {
        return this.length * this.width;
    }
}
public class Main {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(5.5, 3.0);
        System.out.println("Area of Rectangle: " + rectangle.calculateArea());
    }
}
```

SUPER KEYWORD - ACCESSING PARENT CLASS MEMBERS

The super keyword helps in accessing the parent class's methods and constructors.

EXAMPLE PROGRAM:-

```
public class Vehicle {
    public void start() {
        System.out.println("Vehicle is starting");
    }
}
public class Car extends Vehicle {
    @Override
    public void start() {
        super.start();
        System.out.println("Car is starting");
    }
}
public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.start();
    }
}
```

STATIC MEMBERS - SHARED ACROSS INSTANCES

Static members (fields or methods) belong to the class rather than any individual object, meaning they are shared across all instances of the class.

EXAMPLE PROGRAM:-

```
public class BankAccount {
    private String accountHolder;
    private double balance;
    private static int accountCount = 0;
    public BankAccount(String accountHolder, double initialBalance) {
        this.accountHolder = accountHolder;
        this.balance = initialBalance;
        accountCount++;
    }
    public static int getAccountCount() {
        return accountCount;
    }
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: $" + amount);
        } else {
            System.out.println("Deposit amount must be positive");
        }
    }
    public void withdraw(double amount) {
        if (amount <= balance && amount > 0) {
            balance -= amount;
            System.out.println("Withdrawn: $" + amount);
        } else {
            System.out.println("Invalid withdrawal amount");
        }
    }
    public void displayAccountInfo() {
        System.out.println("Account Holder: " + accountHolder + ", Balance: $" + balance);
    }}
public class Main {
    public static void main(String[] args) {
        BankAccount account1 = new BankAccount("John Doe", 1000);
        BankAccount account2 = new BankAccount("Jane Smith", 1500);
        account1.deposit(500);
        account1.withdraw(200);
        account1.displayAccountInfo();
        account2.deposit(700);
        account2.withdraw(300);
        account2.displayAccountInfo();
        System.out.println("Total Accounts Created: " + BankAccount.getAccountCount());
    }
}
```

METHOD REFERENCES - CLEANER CODE USING LAMBDA EXPRESSIONS

Method references provide a cleaner and more concise way of using lambda expressions.

EXAMPLE PROGRAM:-

```
import java.util.Arrays;
import java.util.List;
public class Main {
    public static void printName(String name) {
        System.out.println(name);
    }
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Jane", "Alice", "Bob");

        names.forEach(Main::printName);  // Using method reference
    }
}
```

INNER CLASSES - CLASSES INSIDE ANOTHER CLASS

Inner classes are classes that are nested within another class. They are useful for logically grouping classes that should only be used in the context of the enclosing class.

EXAMPLE PROGRAM:-

```
public class Employee {
    private String name;
    private double salary;
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
    public void createSalarySlip() {
        SalarySlip salarySlip = new SalarySlip();
        salarySlip.printSalaryDetails();
    }
    private class SalarySlip {
        public void printSalaryDetails() {
            System.out.println("Employee: " + name);
            System.out.println("Salary: $" + salary);
        }    }
    public static void main(String[] args) {
        Employee employee = new Employee("Alice", 80000);
        employee.createSalarySlip();
    }}
```

SINGLETON PATTERN - ENSURING A CLASS HAS ONLY ONE INSTANCE

The Singleton pattern ensures that a class has only one instance and provides a global point of access to it.

EXAMPLE PROGRAM:-

```
public class ConfigurationManager {
    private static ConfigurationManager instance;
    private ConfigurationManager() {
        System.out.println("Configuration Manager Initialized");
    }
    public static ConfigurationManager getInstance() {
        if (instance == null) {
            synchronized (ConfigurationManager.class) {
                if (instance == null) {
                    instance = new ConfigurationManager();
                }
            }
        }
        return instance;
    }
    public void loadSetting(String settingName) {
        System.out.println("Loading setting: " + settingName);
    }
    public void displayConfigInfo() {
        System.out.println("Displaying configuration settings...");
    }
}
public class Main {
    public static void main(String[] args) {
        ConfigurationManager configManager = ConfigurationManager.getInstance();
        configManager.loadSetting("Database URL");
        configManager.displayConfigInfo();
        ConfigurationManager anotherConfigManager = ConfigurationManager.getInstance();
        System.out.println("Are both instances the same? " + (configManager ==
anotherConfigManager));
    }}
```

JAVA COLLECTIONS FRAMEWORK OVERVIEW:

- The framework provides a set of classes and interfaces for storing and manipulating data, such as Lists, Sets, Maps, and Queues.
- It helps with efficient data manipulation and iteration, with the root interfaces being **Collection** (for collections of elements) and **Map** (for key-value pairs).

CORE INTERFACES

Collection Interface

The Collection interface is the root for all other collection types. It provides basic functionality for adding, removing, and checking elements.

Key Methods:

1. boolean add(E e): Adds an element to the collection.
2. boolean remove(Object o): Removes the specified element.
3. boolean contains(Object o): Checks if the collection contains the specified element.
4. int size(): Returns the number of elements in the collection.
5. boolean isEmpty(): Checks if the collection is empty.
6. void clear(): Removes all elements from the collection.
7. Object[] toArray(): Converts the collection to an array.

<u>EXAMPLE PROGRAM:</u>

```
import java.util.ArrayList;
import java.util.Collection;
public class CollectionExample {
    public static void main(String[] args) {
        Collection<String> collection = new ArrayList<>();
        collection.add("Apple");
        collection.add("Banana");
        collection.add("Cherry");
        System.out.println("Collection: " + collection); // Output: [Apple, Banana, Cherry]
        collection.remove("Banana");
        System.out.println("After Removal: " + collection); // Output: [Apple, Cherry]
        System.out.println("Contains 'Apple': " + collection.contains("Apple")); // Output: true
        System.out.println("Size: " + collection.size()); // Output: 2
        collection.clear();
        System.out.println("Is Empty: " + collection.isEmpty()); // Output: true
    }
}
```

LIST INTERFACE

The List interface extends Collection and allows ordered collections with duplicate elements. Elements can be accessed via an index.

Implementations:

- ArrayList: Dynamic resizing array.
- LinkedList: Doubly-linked list for fast insertions and deletions.
- Vector: Thread-safe, synchronized version of ArrayList.
- Stack: LIFO implementation of Vector.

Key Methods:

1. void add(int index, E element): Adds an element at the specified index.
2. E get(int index): Returns the element at the specified index.
3. E set(int index, E element): Replaces the element at the specified index.
4. E remove(int index): Removes the element at the specified index.
5. int indexOf(Object o): Returns the index of the first occurrence.
6. int lastIndexOf(Object o): Returns the index of the last occurrence.

EXAMPLE PROGRAM:

```
import java.util.ArrayList;

public class ListExample {
    public static void main(String[] args) {
```

```
    ArrayList<String> list = new ArrayList<>();
    list.add("One");
    list.add("Two");
    list.add("Three");
    list.add(1, "Inserted");
    System.out.println("List: " + list); // Output: [One, Inserted, Two, Three]
    list.remove(2);
    System.out.println("After Removal: " + list); // Output: [One, Inserted, Three]
    System.out.println("Element at index 1: " + list.get(1)); // Output: Inserted
    list.set(0, "Updated");
    System.out.println("After Update: " + list); // Output: [Updated, Inserted, Three]
  }}
```

SET INTERFACE

The Set interface represents a collection of unique elements. It does not allow duplicates.

Implementations:

- HashSet: Backed by a hash table.
- LinkedHashSet: Maintains insertion order.
- TreeSet: Maintains elements in sorted order.

Key Methods:

1. boolean add(E e): Adds the specified element if not already present.
2. boolean remove(Object o): Removes the specified element.
3. boolean contains(Object o): Checks if the element is present.

EXAMPLE PROGRAM:

```
import java.util.HashSet;
```

```
public class SetExample {
   public static void main(String[] args) {
      HashSet<Integer> set = new HashSet<>();
      set.add(10);
      set.add(20);
      set.add(30);
      set.add(10); // Duplicate element
      System.out.println("Set: " + set); // Output: [10, 20, 30]
      set.remove(20);
      System.out.println("After Removal: " + set); // Output: [10, 30]
      System.out.println("Contains 30: " + set.contains(30)); // Output: true
   }}
```

QUEUE INTERFACE

The Queue interface represents a collection designed for holding elements prior to processing. It usually follows FIFO order.

Implementations:

- LinkedList: General-purpose implementation.
- PriorityQueue: Orders elements based on their natural ordering or a comparator.

Key Methods:

1. boolean add(E e): Adds the specified element.
2. E poll(): Retrieves and removes the head of the queue.
3. E peek(): Retrieves the head without removing it.

EXAMPLE PROGRAM:

```
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.add("A");
        queue.add("B");
        queue.add("C");
        System.out.println("Queue: " + queue); // Output: [A, B, C]
        System.out.println("Head: " + queue.peek()); // Output: A
        queue.poll(); // Removes A
        System.out.println("After Poll: " + queue); // Output: [B, C]
    }}
```

MAP INTERFACE

The Map interface maps keys to values. Each key must be unique.

Implementations:

- HashMap: Unordered map.
- LinkedHashMap: Maintains insertion order.
- TreeMap: Sorted map.

Key Methods:

1. V put(K key, V value): Associates the specified value with the key.
2. V get(Object key): Retrieves the value associated with the key.
3. boolean containsKey(Object key): Checks if the map contains the key.
4. boolean containsValue(Object value): Checks if the map contains the value.
5. V remove(Object key): Removes the key-value pair.

EXAMPLE PROGRAM:

```
import java.util.HashMap;
public class MapExample {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();
        map.put(1, "One");
        map.put(2, "Two");
        map.put(3, "Three");
        System.out.println("Map: " + map); // Output: {1=One, 2=Two, 3=Three}
        System.out.println("Value for key 2: " + map.get(2)); // Output: Two
        map.remove(1);
        System.out.println("After Removal: " + map); // Output: {2=Two, 3=Three}
    }}
```

SPECIALIZED METHODS IN COLLECTION IMPLEMENTATIONS

ARRAYLIST

ArrayList is a resizable array implementation of the List interface. It provides random access to elements with dynamic growth.

Key Methods:

1. boolean add(E e): Appends the element.
2. void add(int index, E element): Inserts at the specified index.
3. E get(int index): Retrieves the element.
4. int size(): Returns the number of elements.
5. boolean remove(Object o): Removes the first occurrence.
6. E remove(int index): Removes the element at the index.

EXAMPLE PROGRAM:

```
import java.util.ArrayList;
public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        System.out.println("Fruits: " + fruits); // Output: [Apple, Banana, Cherry]
        fruits.add(1, "Blueberry");
        System.out.println("After Adding: " + fruits); // Output: [Apple, Blueberry, Banana, Cherry]
        fruits.remove("Banana");
        System.out.println("After Removal: " + fruits); // Output: [Apple, Blueberry, Cherry]
    }}
```

LINKEDLIST

LinkedList implements both the List and Deque interfaces. It is a doubly-linked list, suitable for frequent insertions and deletions.

Key Methods (In addition to List methods):

1. void addFirst(E e): Adds an element to the start.
2. void addLast(E e): Adds an element to the end.
3. E getFirst(): Retrieves the first element.
4. E getLast(): Retrieves the last element.
5. E removeFirst(): Removes the first element.
6. E removeLast(): Removes the last element.

EXAMPLE PROGRAM:

```
import java.util.LinkedList;
```

```
public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> animals = new LinkedList<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.addFirst("Elephant");
        animals.addLast("Lion");
        System.out.println("Animals: " + animals); // Output: [Elephant, Dog, Cat, Lion]
        System.out.println("First Animal: " + animals.getFirst()); // Output: Elephant
        System.out.println("Last Animal: " + animals.getLast()); // Output: Lion
        animals.removeFirst();
        System.out.println("After Removing First: " + animals); // Output: [Dog, Cat, Lion]
    }}
```

HASHSET

HashSet implements the Set interface. It uses a hash table for storage, ensuring unique and unordered elements.

Key Methods (Inherited from Set):

1. boolean add(E e): Adds the element if not already present.
2. boolean remove(Object o): Removes the specified element.
3. boolean contains(Object o): Checks for the presence of an element.

EXAMPLE PROGRAM:

```
import java.util.HashSet;
public class HashSetExample {
    public static void main(String[] args) {
        HashSet<Integer> numbers = new HashSet<>();
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);
        System.out.println("HashSet: " + numbers); // Output: [10, 20, 30]
        numbers.add(20); // Duplicate element, no effect
        System.out.println("After Adding Duplicate: " + numbers); // Output: [10, 20, 30]
        numbers.remove(10);
        System.out.println("After Removal: " + numbers); // Output: [20, 30]
    }
}
```

<u>TREESET</u>

TreeSet implements the Set interface and maintains elements in sorted order. It is backed by a TreeMap.

Key Methods (Inherited from NavigableSet):

1. E first(): Returns the first (lowest) element.
2. E last(): Returns the last (highest) element.
3. E ceiling(E e): Returns the least element ≥ e.
4. E floor(E e): Returns the greatest element ≤ e.

<u>EXAMPLE PROGRAM:</u>

```
import java.util.TreeSet;
public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(50);
        numbers.add(20);
        numbers.add(40);
        numbers.add(10);
        System.out.println("TreeSet: " + numbers); // Output: [10, 20, 40, 50]
        System.out.println("First: " + numbers.first()); // Output: 10
        System.out.println("Last: " + numbers.last()); // Output: 50
        System.out.println("Ceiling of 25: " + numbers.ceiling(25)); // Output: 40
        System.out.println("Floor of 25: " + numbers.floor(25)); // Output: 20
    }
}
```

PRIORITYQUEUE

PriorityQueue implements the Queue interface, ordering elements according to their natural order or a custom comparator.

Key Methods:

1. boolean offer(E e): Inserts the element.
2. E poll(): Retrieves and removes the head of the queue.
3. E peek(): Retrieves the head without removing it.

EXAMPLE PROGRAM:

```
import java.util.PriorityQueue;
```

```
public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        pq.offer(30);
        pq.offer(20);
        pq.offer(10);

        System.out.println("PriorityQueue: " + pq); // Output: [10, 30, 20]
        System.out.println("Peek: " + pq.peek()); // Output: 10

        pq.poll(); // Removes 10
        System.out.println("After Poll: " + pq); // Output: [20, 30]
    }
}
```

ADVANCED FEATURES

4.1 Comparator Interface

The Comparator interface defines a custom ordering for collections.

EXAMPLE PROGRAM:

```
import java.util.*;
class Person {
    String name;
    int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }}
public class ComparatorExample {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));
        // Sort by age
        people.sort(Comparator.comparingInt(p -> p.age));
        for (Person p : people) {
            System.out.println(p.name + " - " + p.age);
        }       // Output:
        // Bob - 25
        // Alice - 30
        // Charlie - 35    }}
```

ITERATOR INTERFACE

The Iterator interface allows traversing collections.

Key Methods:

1. boolean hasNext(): Checks if more elements exist.
2. E next(): Returns the next element.
3. void remove(): Removes the last returned element.

EXAMPLE PROGRAM:

```
import java.util.ArrayList;
import java.util.Iterator;
```

```
public class IteratorExample {
   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList<>();
      list.add("A");
      list.add("B");
      list.add("C");
      Iterator<String> it = list.iterator();
      while (it.hasNext()) {
         System.out.println(it.next());
      }
      // Output: A B C
   }
}
```

STREAMS

Streams provide functional-style operations for collections.

Key Methods:

1. filter(): Filters elements based on a predicate.
2. map(): Transforms elements.
3. forEach(): Performs an action on each element.

EXAMPLE PROGRAM:

```
import java.util.Arrays;
import java.util.List;
public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        numbers.stream()
               .filter(n -> n % 2 == 0)
               .forEach(System.out::println);
        // Output: 2 4
    }
}
```

MAP HIERARCHY IN JAVA

The Map interface in Java is part of the Collections Framework and is used to store key-value pairs. It does not inherit from the Collection interface but is a separate hierarchy.

Hierarchy of Map

1. **Map Interface**
   - Represents a mapping between a key and a value.
2. **HashMap**
   - Implements the Map interface using a hash table.
3. **LinkedHashMap**
   - Extends HashMap and maintains insertion order.
4. **TreeMap**
   - Implements NavigableMap and maintains elements in a sorted order.
5. **WeakHashMap**

- A map with keys eligible for garbage collection.

6. **IdentityHashMap**
    - Uses reference equality instead of equals() for comparing keys.
7. **ConcurrentHashMap**
    - A thread-safe implementation for high concurrency.
8. **EnumMap**
    - A specialized map for keys of enum type.

Key Methods in the Map Interface

General Methods

1. V put(K key, V value): Associates the specified value with the key.
2. V get(Object key): Returns the value associated with the key.
3. V remove(Object key): Removes the key-value pair for the key.
4. boolean containsKey(Object key): Checks if the key is present.
5. boolean containsValue(Object value): Checks if the value is present.
6. int size(): Returns the number of key-value mappings.
7. boolean isEmpty(): Checks if the map is empty.
8. void clear(): Removes all mappings.
9. Set<K> keySet(): Returns a set of all keys.
10. Collection<V> values(): Returns a collection of all values.
11. Set<Map.Entry<K, V>> entrySet(): Returns a set of all key-value mappings.

Additional Methods (NavigableMap)

1. K firstKey(): Returns the first key.
2. K lastKey(): Returns the last key.
3. K ceilingKey(K key): Returns the least key ≥ given key.

4. K floorKey(K key): Returns the greatest key ≤ given key.

HASHMAP

HashMap is a non-synchronized map that uses hash codes for storage.

EXAMPLE PROGRAM:

```
import java.util.HashMap;
public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("A", 1);
        map.put("B", 2);
        map.put("C", 3);
        System.out.println("HashMap: " + map); // Output: {A=1, B=2, C=3}
        System.out.println("Value for key 'B': " + map.get("B")); // Output: 2
        map.remove("A");
        System.out.println("After Removing 'A': " + map); // Output: {B=2, C=3}
        System.out.println("Contains Key 'B': " + map.containsKey("B")); // Output: true
        System.out.println("Contains Value 3: " + map.containsValue(3)); // Output: true
    }
}
```

LINKEDHASHMAP

LinkedHashMap maintains the insertion order.

EXAMPLE PROGRAM:

```
import java.util.LinkedHashMap;
public class LinkedHashMapExample {
    public static void main(String[] args) {
        LinkedHashMap<String, Integer> map = new LinkedHashMap<>();
        map.put("Apple", 50);
        map.put("Banana", 20);
        map.put("Cherry", 30);
        System.out.println("LinkedHashMap: " + map); // Output: {Apple=50, Banana=20, Cherry=30}
        map.put("Banana", 25); // Updates value for key 'Banana'
        System.out.println("After Update: " + map); // Output: {Apple=50, Banana=25, Cherry=30}
    }
}
```

TREEMAP

TreeMap maintains keys in sorted order.

EXAMPLE PROGRAM:

```
import java.util.TreeMap;
public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> map = new TreeMap<>();
        map.put("C", 3);
        map.put("A", 1);
        map.put("B", 2);
        System.out.println("TreeMap: " + map); // Output: {A=1, B=2, C=3}
        System.out.println("First Key: " + map.firstKey()); // Output: A
        System.out.println("Last Key: " + map.lastKey()); // Output: C
    }
}
```

WEAKHASHMAP

WeakHashMap allows garbage collection of keys that are no longer referenced.

EXAMPLE PROGRAM:

```
import java.util.WeakHashMap;
public class WeakHashMapExample {
    public static void main(String[] args) {
        WeakHashMap<String, String> map = new WeakHashMap<>();
        String key = new String("Key");
        map.put(key, "Value");
        System.out.println("Before GC: " + map); // Output: {Key=Value}
        key = null;
        System.gc(); // Garbage collection
        System.out.println("After GC: " + map); // Output: {}
    }
}
```

IDENTITYHASHMAP

IdentityHashMap compares keys using == instead of equals().

EXAMPLE PROGRAM:

```
import java.util.IdentityHashMap;
public class IdentityHashMapExample {
    public static void main(String[] args) {
        IdentityHashMap<String, String> map = new IdentityHashMap<>();
        String key1 = new String("Key");
        String key2 = new String("Key");
        map.put(key1, "Value1");
        map.put(key2, "Value2");
        System.out.println("IdentityHashMap: " + map);
        // Output: {Key=Value1, Key=Value2}
    }}
```

CONCURRENTHASHMAP

ConcurrentHashMap provides thread-safe access to key-value mappings.

EXAMPLE PROGRAM:

```
import java.util.concurrent.ConcurrentHashMap;
public class ConcurrentHashMapExample {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
        map.put("One", 1);
        map.put("Two", 2);
        map.forEach((key, value) -> System.out.println(key + ": " + value));
        // Output: One: 1
        //         Two: 2
    }}
```

ENUMMAP

EnumMap is a specialized map for keys of an enum type.

EXAMPLE PROGRAM:

```
import java.util.EnumMap;
enum Days {
   MONDAY, TUESDAY, WEDNESDAY
}
public class EnumMapExample {
   public static void main(String[] args) {
      EnumMap<Days, String> map = new EnumMap<>(Days.class);
      map.put(Days.MONDAY, "Work");
      map.put(Days.TUESDAY, "Gym");
      System.out.println("EnumMap: " + map); // Output: {MONDAY=Work,
TUESDAY=Gym}
   }
}
```

ENTRY INTERFACE

The Map.Entry interface represents a single key-value pair in the map.

EXAMPLE PROGRAM:

```
import java.util.HashMap;
import java.util.Map;
public class MapEntryExample {
   public static void main(String[] args) {
      Map<String, Integer> map = new HashMap<>();
```

```
        map.put("A", 1);
        map.put("B", 2);
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
        // Output: A: 1
        //         B: 2
    }
}
```

COLLECTIONS :

The Collection interface in Java provides a variety of methods to manipulate and interact with collections. Below is a summary of the most commonly used methods in the Collection interface, along with a brief explanation of each:

1. Adding Elements

- add(E e): Adds the specified element to the collection.
  - Example: list.add("Apple");
- addAll(Collection<? extends E> c): Adds all elements from the specified collection to the collection.
  - Example: list.addAll(anotherList);

2. Removing Elements

- remove(Object o): Removes the specified element from the collection if it exists.
  - Example: list.remove("Apple");
- removeAll(Collection<?> c): Removes all elements in the collection that are also contained in the specified collection.
  - Example: list.removeAll(anotherList);

- retainAll(Collection<?> c): Retains only the elements in the collection that are contained in the specified collection.
    - Example: list.retainAll(anotherList);
- clear(): Removes all elements from the collection.
    - Example: list.clear();

3. Checking Properties

- size(): Returns the number of elements in the collection.
    - Example: int size = list.size();
- isEmpty(): Returns true if the collection contains no elements.
    - Example: boolean empty = list.isEmpty();
- contains(Object o): Returns true if the collection contains the specified element.
    - Example: boolean hasApple = list.contains("Apple");
- containsAll(Collection<?> c): Returns true if the collection contains all elements in the specified collection.
    - Example: boolean hasAll = list.containsAll(anotherList);
- toArray(): Returns an array containing all the elements in the collection.
    - Example: Object[] array = list.toArray();

4. Iteration

- iterator(): Returns an iterator over the elements in the collection.
    - Example:

```
java
Copy code
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
   System.out.println(iterator.next());
}
```

- forEach(Consumer<? super E> action): Performs the given action for each element of the collection.
  - Example: list.forEach(System.out::println);

<u>MULTITHREADING IN JAVA</u>

1. Thread in Java

A **thread** in Java represents a single execution path within a program. The Java programming language allows multiple threads to run concurrently, making it ideal for performing multiple tasks at once. Each thread operates independently, but they share the same memory space (heap), which allows them to communicate more easily than processes.

Java threads are lightweight, as they are managed by the Java Virtual Machine (JVM) and can be executed in parallel by utilizing multiple CPU cores. Java provides two main ways to create threads:

- **Extending the Thread class**: A class extends Thread and overrides the run() method.
- **Implementing the Runnable interface**: A class implements Runnable and provides the implementation of the run() method.

Java threads run concurrently, but not necessarily in parallel unless you have multiple processors. The JVM determines the scheduling of threads, which makes Java an excellent choice for concurrent applications.

2. Thread Life Cycle

The thread life cycle in Java defines the sequence of states that a thread goes through during its existence. The states include:

1. **New (Born)**:

    - A thread is in the "New" state right after it is created, but before the start() method is called.
    - In this state, no CPU resources are allocated yet.
2. **Runnable**:
    - After the thread is started (via the start() method), it enters the "Runnable" state.
    - The thread can run at any point in time but is not guaranteed immediate execution. It waits for the JVM to allocate CPU resources.
    - The thread remains in the runnable state until it gets CPU time for execution.
3. **Blocked**:
    - A thread enters the "Blocked" state when it cannot proceed because another thread is holding a lock on a shared resource.
    - This typically occurs when one thread is waiting to enter a synchronized block while another thread is already inside it.
4. **Waiting**:
    - A thread enters the "Waiting" state when it is waiting for another thread to perform a particular action.
    - This could occur when a thread calls methods like wait(), sleep(), or join().
5. **Terminated (Dead)**:
    - The thread enters the "Terminated" state when its run() method completes or it is explicitly terminated.
    - A dead thread cannot be restarted.

3. Thread Priority

Thread priority in Java allows you to influence the order in which threads are executed. The JVM typically uses thread priority values to prioritize the execution of threads.

Java provides three priority levels:

- **High priority**: Thread.MAX_PRIORITY (value = 10)

- **Normal priority**: Thread.NORM_PRIORITY (value = 5, default)
- **Low priority**: Thread.MIN_PRIORITY (value = 1)

The JVM scheduler uses these priorities to allocate CPU time to threads, but it is not guaranteed that higher-priority threads will always be executed before lower-priority ones. This depends on the underlying OS's thread scheduling algorithm.

When to use Thread Priority:

- **Time-sensitive tasks**: High-priority threads are useful for tasks that need to be executed without delay (e.g., real-time data processing).
- **Background tasks**: Low-priority threads are suitable for non-critical tasks that can wait for resources.

4. Thread Class

The Thread class in Java is a built-in class that allows you to create and manage threads. Some important methods in the Thread class include:

- **start()**: This method is used to begin the execution of a thread. Once start() is called, the thread enters the "Runnable" state.
- **run()**: This method contains the logic to be executed by the thread. It is called automatically when a thread starts.
- **sleep(long millis)**: Pauses the thread for the specified number of milliseconds.
- **setPriority(int priority)**: This method sets the priority of a thread. The priority is a value between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY.
- **join()**: Makes the current thread wait until the thread it is called on has completed its execution.

Why Use the Thread Class:

- It is useful when you want to directly control the execution of a thread and need custom implementations of the run() method.

5. Runnable Interface

The Runnable interface is a functional interface that represents a task to be executed by a thread. It provides a more flexible way of creating threads compared to extending the Thread class.

Instead of creating a subclass of Thread, you implement the Runnable interface and provide the implementation of the run() method. This approach allows your class to extend other classes as well, which would not be possible if it extended Thread.

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread is running.");
    }
}
```

Benefits:

- It provides better flexibility since a class can implement Runnable and still extend another class.
- It decouples the task from the thread management logic, making the task more reusable.

6. Synchronized Methods

In multithreading, **synchronization** is crucial for preventing data corruption when multiple threads access shared resources. Without synchronization, one thread might read or write to a shared variable while another thread is also modifying it, leading to inconsistencies.

By marking a method with the synchronized keyword, Java ensures that only one thread can execute it at a time for each instance of the class.

```
public synchronized void increment() {
    counter++;
}
```

When to use synchronization:

- **Data Integrity**: When multiple threads are modifying the same resource (e.g., counters, lists).
- **Critical Sections**: When you want to ensure that a piece of code that modifies shared resources is executed by only one thread at a time.

7. Daemon Threads

A **daemon thread** is a special type of thread that runs in the background and is used to perform tasks such as garbage collection, event handling, or system monitoring.

Daemon threads do not prevent the JVM from shutting down when all user threads (non-daemon threads) are completed.

Creating Daemon Threads: Daemon threads are created by calling setDaemon(true) before starting the thread.

```
Thread daemonThread = new Thread(new Runnable() {
    public void run() {
        // Task executed by daemon thread
    }
});
daemonThread.setDaemon(true);
daemonThread.start();
```

When to use daemon threads:

- Use daemon threads for tasks that should not block program termination, such as background processing or monitoring tasks.

8. Thread Pool

A **thread pool** is a collection of threads that are used to execute tasks. Instead of creating new threads for each task, you use a pool of pre-existing threads to execute multiple tasks. This improves the efficiency of your program by minimizing the overhead associated with creating and destroying threads.

The ExecutorService framework provides thread pool management capabilities. Common thread pool types are:

- **FixedThreadPool**: A thread pool with a fixed number of threads.
- **CachedThreadPool**: A thread pool that creates new threads as needed, but will reuse existing ones when possible.
- **ScheduledThreadPool**: A thread pool for scheduled tasks (similar to cron jobs).

```
ExecutorService pool = Executors.newFixedThreadPool(5);
pool.submit(new Task());
pool.shutdown();
```

Why use Thread Pools:

- **Efficiency**: Reduces the cost of thread creation and destruction.
- **Management**: Makes it easier to manage the number of concurrent threads.
- **Scalability**: Can handle a large number of tasks efficiently by using a small number of threads.

9. Thread Group

A **Thread Group** is a way to organize and manage related threads. A thread group allows you to perform collective actions on all threads within that group, such as changing their priorities or interrupting them.

You can use thread groups to manage threads with similar behavior or to separate threads logically for different purposes.

```
ThreadGroup group = new ThreadGroup("Group1");
Thread t1 = new Thread(group, new Runnable() {
    public void run() {
        System.out.println("Task 1");
    }
});
```

When to use Thread Groups:

- When you want to apply an operation on multiple threads (e.g., interrupting a group of threads simultaneously).
- When you want to organize threads in a way that reflects their purpose or lifecycle.

EXAMPLE PROGRAM

```
import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

class ShoppingCart {

    private final String customerName;

    private final int[] itemPrices;

    private int totalBill = 0;
    public ShoppingCart(String customerName, int[] itemPrices) {

        this.customerName = customerName;

        this.itemPrices = itemPrices;

    }

    public synchronized void calculateTotalBill() {

        System.out.println("Calculating total for " + customerName + "...");

        for (int price : itemPrices) {

            totalBill += price;

        }

        System.out.println(customerName + "'s total bill: ₹" + totalBill);

    }

    public String getCustomerName() {

        return customerName;

    }

    public int getTotalBill() {

        return totalBill;

    }}

class CustomerCheckout implements Runnable {
    private final ShoppingCart cart;

    public CustomerCheckout(ShoppingCart cart) {

        this.cart = cart;

    }

    @Override

    public void run() {

        cart.calculateTotalBill();

        System.out.println("Checkout completed for " + cart.getCustomerName());

    }}

public class Store {

    public static void main(String[] args) {

        ShoppingCart[] carts = {

            new ShoppingCart("Arun", new int[]{150, 300, 450}),

            new ShoppingCart("Lakshmi", new int[]{100, 200, 250}),

            new ShoppingCart("Vijay", new int[]{500, 600}),

            new ShoppingCart("Priya", new int[]{50, 75, 125}),

            new ShoppingCart("Meena", new int[]{1000, 2000})

        };
        ExecutorService executorService = Executors.newFixedThreadPool(3);

        for (ShoppingCart cart : carts) {

            executorService.submit(new CustomerCheckout(cart));

        }

        executorService.shutdown();

    }}
```

OUTPUT:-

```
Calculating total for Arun...
Calculating total for Lakshmi...
Lakshmi's total bill: ₹550
Calculating total for Vijay...
Arun's total bill: ₹900
Checkout completed for Lakshmi
Calculating total for Priya...
Priya's total bill: ₹250
Checkout completed for Priya
Calculating total for Meena...
Meena's total bill: ₹3000
Checkout completed for Meena
Checkout completed for Arun
Vijay's total bill: ₹1100

Checkout completed for Vijay
```

WHAT IS SQL?

SQL (Structured Query Language) is the standard programming language used to manage and manipulate relational databases. It allows for querying, inserting, updating, and deleting data. SQL commands are used for defining, managing, and querying data in a relational database.

WHAT IS A DATABASE?

A **database** is a structured collection of data. It is designed to store, manage, and retrieve large amounts of data efficiently. Databases are commonly managed using a **Database Management System (DBMS)**. Popular DBMS software includes MySQL, PostgreSQL, SQL Server, and Oracle.

WHAT IS MYSQL DATABASE?

MySQL is an open-source, relational database management system that uses SQL for managing data. It is widely used in web applications due to its performance, reliability, and ease of use. It supports a variety of data types and is capable of handling large volumes of data.

TYPES OF SQL COMMANDS

SQL commands can be classified into several categories:

1. **Data Query Language (DQL)**:
    - **SELECT**: Used to query data from a database.
2. **Data Definition Language (DDL)**:
    - **CREATE**: Used to create database objects like tables.
    - **ALTER**: Used to modify database objects.
    - **DROP**: Used to delete database objects.
3. **Data Manipulation Language (DML)**:
    - **INSERT**: Used to insert data into tables.
    - **UPDATE**: Used to modify existing data.
    - **DELETE**: Used to delete data from tables.
4. **Data Control Language (DCL)**:

- **GRANT**: Used to give permissions.
- **REVOKE**: Used to remove permissions.

CONSTRAINTS IN SQL

Constraints are used to ensure data integrity and consistency in a relational database. The most common types of constraints are:

- **NOT NULL**: Ensures a column does not accept NULL values.
- **UNIQUE**: Ensures all values in a column are unique.
- **PRIMARY KEY**: Uniquely identifies each row in a table.
- **FOREIGN KEY**: Establishes a relationship between two tables.
- **CHECK**: Ensures values meet a certain condition.
- **DEFAULT**: Provides a default value for a column.

SQL OPERATORS

SQL operators allow you to perform operations on data within queries. Here are some common operators:

1. **Arithmetic Operators**:
   - +, -, *, /, % (addition, subtraction, multiplication, division, modulus).
2. **Comparison Operators**:
   - =, !=, >, <, >=, <= (equal, not equal, greater than, less than, etc.).
3. **Logical Operators**:

    - AND, OR, NOT.
4. **BETWEEN**: Filters values within a given range.
5. **IN**: Checks if a value matches any value in a list.
6. **LIKE**: Allows pattern matching.
7. **IS NULL**: Checks for NULL values.

Example of Arithmetic Operators

```
SELECT salary * 1.10 AS new_salary
FROM employees;
```

Output:

```
new_salary
55000.00
62000.00
72000.00
```

This query increases each employee's salary by 10%.

Example of Comparison Operators

```
SELECT name, salary
FROM employees
WHERE salary > 50000;
```

Output:

```
name salary
John   60000
```

```
name salary
Sarah 75000
```

This query retrieves employees whose salary is greater than 50,000.

Example of Logical Operators

```
SELECT name, salary
FROM employees
WHERE salary > 50000 AND dept_id = 101;
```

Output:

```
name salary
John   60000
```

This query retrieves employees who earn more than 50,000 and work in department 101.

Example of LIKE Operator

```
SELECT name
FROM employees
WHERE name LIKE 'J%';
```

Output:

name
John
James

This query retrieves employees whose names start with the letter 'J'.

STORING BLOB DATA

A **BLOB** (Binary Large Object) is used to store large binary data such as images, videos, or audio files. In MySQL, BLOB types allow you to store large amounts of binary data.

Example of Storing BLOB Data

```
CREATE TABLE images (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    image_data BLOB
);
INSERT INTO images (name, image_data)
VALUES ('Sample Image', LOAD_FILE('/path/to/image.jpg'));
```

Output:

The image is stored in the image_data column of the table. It is now saved as a BLOB in the database.

RETRIEVING BLOB DATA

To retrieve the image data, you would use the following query:

```
SELECT name, image_data
FROM images
WHERE id = 1;
```

Output:

This query fetches the name of the image and the binary data (image) stored in the image_data column.

WHAT IS A VIEW IN SQL?

A **view** in SQL is a virtual table created by a query that selects data from one or more tables. Views do not store data physically but provide a dynamic way of presenting data based on specific queries.

Creating a View

```
CREATE VIEW EmployeeView AS
SELECT name, salary, dept_id
FROM employees
WHERE salary > 50000;
```

Output:

name	salary	dept_id
John	60000	101
Sarah	75000	102

This view shows the names, salaries, and department IDs of employees earning more than 50,000.

Using a View

```
SELECT * FROM EmployeeView;
```

Output:

```
name salary dept_id
John   60000 101
Sarah 75000 102
```

This query retrieves all data from the EmployeeView.

WHAT IS A TRIGGER IN SQL?

A **trigger** is a set of SQL statements that are automatically executed (or "fired") when a specified event occurs on a table or view. It can be set to run **before** or **after** events such as INSERT, UPDATE, or DELETE.

Before INSERT Trigger

```
CREATE TRIGGER before_insert_employee
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
```

```
    SET NEW.salary = ROUND(NEW.salary, 2);
END;
```

Output:

Before a new employee's salary is inserted, it will be rounded to two decimal places.

After UPDATE Trigger

```
CREATE TRIGGER after_update_employee
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO employee_audit (emp_id, old_salary, new_salary)
    VALUES (NEW.emp_id, OLD.salary, NEW.salary);
END;
```

Output:
After an employee's salary is updated, the change is logged in the employee_audit table.

WHAT IS A FOREIGN KEY?

A **Foreign Key (FK)** is a column or a set of columns in one table that refers to the primary key of another table. The foreign key establishes a relationship between two tables and ensures referential integrity. This means that the data in the foreign key column must correspond to values in the primary key column of the referenced table, or it can be null.

Example of Foreign Key Constraint

Consider two tables: orders and customers. We want to establish a relationship between these two tables using the customer_id column in the orders table, which refers to the id column in the customers table.

```
CREATE TABLE customers (
   id INT PRIMARY KEY,
   name VARCHAR(100),
   email VARCHAR(100)
);
CREATE TABLE orders (
   order_id INT PRIMARY KEY,
   order_date DATE,
   amount DECIMAL(10, 2),
   customer_id INT,
   FOREIGN KEY (customer_id) REFERENCES customers(id)
);
```

Output:

The orders table now references the customers table using the customer_id column. When inserting data into the orders table, the value in customer_id must exist in the customers table, ensuring referential integrity.

WHAT IS A JOIN IN SQL?A **Join** is used to combine rows from two or more tables based on a related column between them. There are several types of joins, including:

- **INNER JOIN**: Returns records that have matching values in both tables.
- **LEFT JOIN** (or LEFT OUTER JOIN): Returns all records from the left table and matched records from the right table, or NULL if there is no match.
- **RIGHT JOIN** (or RIGHT OUTER JOIN): Returns all records from the right table and matched records from the left table, or NULL if there is no match.
- **FULL JOIN** (or FULL OUTER JOIN): Returns records when there is a match in either left or right table.
- **CROSS JOIN**: Returns the Cartesian product of both tables (every combination of rows from both tables).

Example of INNER JOIN

```
SELECT customers.name, orders.order_id, orders.amount
FROM customers
INNER JOIN orders ON customers.id = orders.customer_id;
```

Output:

name	order_id	amount
John	101	250.00
Sarah	102	500.00

This query retrieves the customer name, order ID, and order amount for all customers who have placed an order, using the INNER JOIN between customers and orders.

Example of LEFT JOIN

```
SELECT customers.name, orders.order_id, orders.amount
FROM customers
```

```
LEFT JOIN orders ON customers.id = orders.customer_id;
```

Output:

name	order_id	amount
John	101	250.00
Sarah	102	500.00
Mike	NULL	NULL

This query retrieves all customers, including those who have not placed an order. If no matching records are found in the orders table, the result is NULL.

WHAT IS AN INDEX IN SQL?

An **Index** is a database object that improves the speed of data retrieval operations on a table at the cost of additional space and maintenance time. Indexes are often created on columns that are frequently used in WHERE, JOIN, or ORDER BY clauses.

There are two common types of indexes:

1. **Primary Index**: Automatically created on the primary key column.
2. **Secondary Index**: Manually created on other columns to improve query performance.

Creating an Index

```
 CREATE INDEX idx_email ON customers (email);
```

Output:

The idx_email index is created on the email column of the customers table. This index will speed up queries that filter or sort by the email column.

WHAT IS DATA INTEGRITY?

Data integrity refers to the accuracy, consistency, and reliability of data stored in a database. It ensures that data remains accurate and consistent throughout its lifecycle. The primary mechanisms for maintaining data integrity are constraints, such as NOT NULL, UNIQUE, PRIMARY KEY, and FOREIGN KEY.

Example of Maintaining Data Integrity with Constraints

```
CREATE TABLE employees (
   emp_id INT PRIMARY KEY,
   name VARCHAR(100) NOT NULL,
   department_id INT,
   salary DECIMAL(10, 2) CHECK (salary > 0),
   FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

Output:

The table ensures that:

- The emp_id is unique and not null (using PRIMARY KEY).
- The name cannot be empty (using NOT NULL).

- The salary must be greater than zero (using CHECK).
- The department_id exists in the departments table (using FOREIGN KEY).

WHAT IS A SUBQUERY IN SQL?

A **subquery** is a query nested inside another query. Subqueries are used to retrieve intermediate results that are then used in the main query. A subquery can be used in the SELECT, INSERT, UPDATE, or DELETE statements.

Example of a Subquery

```
 SELECT name
FROM employees
WHERE department_id = (
    SELECT department_id
    FROM departments
    WHERE dept_name = 'HR'
);
```

Output:

```
name
John
Mike
```

This query retrieves the names of employees who work in the "HR" department. The subquery fetches the department_id for "HR", and the main query uses it to filter employees.

WHAT IS A STORED PROCEDURE IN SQL?

A **Stored Procedure** is a set of SQL statements that are stored in the database and can be executed as a single unit. Stored procedures are used to encapsulate logic, improve performance, and reduce redundancy.

Creating a Stored Procedure

```
CREATE PROCEDURE GetEmployeeDetails(IN emp_id INT)
BEGIN
    SELECT name, salary
    FROM employees
    WHERE emp_id = emp_id;
 END;
```

Output:

The stored procedure GetEmployeeDetails takes an employee ID as an input parameter and retrieves the name and salary of the employee with that ID.

Executing a Stored Procedure

```
CALL GetEmployeeDetails(1);
```

Output:

```
name salary
John   60000
```

This query executes the GetEmployeeDetails procedure and returns the details of the employee with emp_id = 1.

WHAT IS A TRANSACTION IN SQL?

A **Transaction** is a set of SQL operations that are executed as a single unit. A transaction ensures that all operations within it are completed successfully before committing the changes. If any operation fails, the transaction can be rolled back to maintain consistency.

Transaction Commands

- **START TRANSACTION**: Begins a new transaction.
- **COMMIT**: Saves the changes made in the transaction.
- **ROLLBACK**: Reverts any changes made during the transaction if an error occurs.

Example of a Transaction

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT;
```

Output:

This transaction transfers 100 units from one account to another. If both updates succeed, the changes are committed. If any update fails, a ROLLBACK can be issued to undo the changes.

WHAT IS A BACKUP IN SQL?

A **Backup** is a copy of the database that can be used to restore data in case of system failure. SQL databases provide several ways to create backups, such as using SQL commands or DBMS tools.

BACKING UP A MYSQL DATABASE

```
 my dump -u root -p database_name > backup.
```

Output:

This command creates a backup of the database_name and saves it in the backup. file.

RESTORING A MYSQL DATABASE

```
my  -u root -p database_name < backup.
```

Output:

This command restores the database from the backup. file.

Sure! Here's the continuation:

WHAT IS A TRIGGER IN SQL?

A **Trigger** is a special type of stored procedure that automatically runs when certain events occur on a particular table or view. Triggers are commonly used to enforce data integrity, automate system functions, and maintain audit logs. Triggers can be set to execute after (AFTER), before (BEFORE), or instead of (INSTEAD OF) a specified database operation, such as INSERT, UPDATE, or DELETE.

Creating a Trigger

Consider a scenario where you want to automatically update an audit table whenever a record is inserted into the employees table. Here's an example of creating a trigger for this purpose:

```
CREATE TRIGGER employee_insert_trigger
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
   INSERT INTO audit_log (action, table_name, timestamp)
   VALUES ('INSERT', 'employees', NOW());
END;
```

Output:

The employee_insert_trigger trigger automatically inserts a record into the audit_log table every time a new employee record is inserted into the employees table. The log includes the action (INSERT), table name, and timestamp of the operation.

<u>WHAT IS A VIEW IN SQL?</u>

A **View** is a virtual table that is derived from one or more base tables. Views simplify complex queries, provide an abstraction layer, and can be used to enhance security by exposing only specific columns or rows from underlying tables. Views do not store data themselves but rather store the SQL query that retrieves the data.

Creating a View

Here’s an example of creating a view that provides a summary of employee salaries by department:

```
CREATE VIEW department_salary_summary AS
SELECT department_id, SUM(salary) AS total_salary
FROM employees
GROUP BY department_id;
```

Output:

The department_salary_summary view provides the total salary of employees for each department. To retrieve the data, you can query the view as you would a regular table:

```
SELECT * FROM department_salary_summary;
```

Result:

department_id	total_salary
1	120000
2	85000

WHAT ARE AGGREGATE FUNCTIONS IN SQL?

Aggregate functions perform a calculation on a set of values and return a single value. Common aggregate functions include:

- COUNT(): Returns the number of rows in a set.
- SUM(): Returns the sum of a numeric column.
- AVG(): Returns the average value of a numeric column.

- MIN(): Returns the smallest value in a set.
- MAX(): Returns the largest value in a set.

Example of Aggregate Functions

```
SELECT COUNT(*) AS total_employees FROM employees;
SELECT AVG(salary) AS average_salary FROM employees;
SELECT MAX(salary) AS highest_salary FROM employees;
```

Output:

total_employees	average_salary	highest_salary
100	55000	100000

These queries calculate the total number of employees, average salary, and highest salary in the employees table.

WHAT IS A BLOB IN SQL?

A **BLOB** (Binary Large Object) is a data type used to store large binary data such as images, audio, and video. BLOBs can store up to several gigabytes of data, depending on the database system.

Storing and Retrieving a BLOB

To store a BLOB in MySQL, you can use the BLOB data type for the column, and for inserting an image, you would typically use a prepared statement to handle the binary data.

```
 CREATE TABLE product_images (
    product_id INT PRIMARY KEY,
    image BLOB
);
-- To insert an image:
INSERT INTO product_images (product_id, image)
VALUES (1, LOAD_FILE('/path/to/image.jpg'));
```

Output:

The image is stored in the image column of the product_images table. To retrieve the image, you can query the table and return the BLOB data.

WHAT IS A REGULAR EXPRESSION IN SQL?

A **Regular Expression (regex)** is a sequence of characters that defines a search pattern. In SQL, regular expressions are used for pattern matching within string data, providing more advanced search capabilities than the LIKE operator.

Example of Regular Expression

In MySQL, the REGEXP operator allows you to search for patterns in string columns.

```
SELECT * FROM employees WHERE name REGEXP '^A';
```

Output:

This query retrieves all employees whose names start with the letter "A."

WORKING WITH SQL SUBQUERIES : A subquery is a query nested inside another query. Subqueries are commonly used to break down complex queries into simpler components. They allow for filtering, aggregation, and other operations based on data that is derived from another query.

Types of Subqueries

There are different types of subqueries, such as:

1. **Single-row Subquery**: Returns a single value.
2. **Multi-row Subquery**: Returns multiple values.
3. **Correlated Subquery**: A subquery that references columns from the outer query.
4. **Non-correlated Subquery**: A subquery that does not reference columns from the outer query.

Example of a Single-row Subquery

A single-row subquery returns a single value. For example, to retrieve employees whose salary is higher than the average salary in the company:

```
SELECT * FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

Output:

This query returns all employees whose salary exceeds the average salary of all employees in the company.

Example of a Multi-row Subquery

A multi-row subquery returns multiple values. For example, you can use a multi-row subquery to get employees who work in departments that have more than 10 employees:

```
SELECT * FROM employees
WHERE department_id IN (SELECT department_id FROM employees GROUP BY department_id HAVING COUNT(*) > 10);
```

Output:

This query returns employees who belong to departments with more than 10 employees.

Example of a Correlated Subquery

A correlated subquery refers to columns from the outer query. For instance, to find employees who earn more than the average salary in their respective department:

```
SELECT * FROM employees e
WHERE salary > (SELECT AVG(salary) FROM employees WHERE department_id = e.department_id);
```

Output:

This query returns employees whose salary is greater than the average salary of their department.

WHAT ARE INDEXES IN SQL?

An **index** is a database object that speeds up the retrieval of rows from a table. Indexes are particularly useful when dealing with large datasets as they reduce the amount of data that must be scanned during a query. Indexes are created on one or more columns of a table.

Creating an Index

Here's an example of creating an index on the employee_id column in the employees table:

```
CREATE INDEX idx_employee_id ON employees(employee_id);
```

Output:

The index idx_employee_id is created on the employee_id column, which will improve the performance of queries that filter or sort by employee_id.

Types of Indexes

There are various types of indexes, including:

1. **Unique Index**: Ensures that all values in the indexed column are unique.
2. **Primary Index**: Automatically created when you define a primary key.
3. **Composite Index**: Created on multiple columns.
4. **Full-text Index**: Used for searching large text data.

Example of a Composite Index

To improve the performance of queries filtering on both first_name and last_name columns, you can create a composite index:

```
CREATE INDEX idx_name ON employees(first_name, last_name);
```

Output:

The composite index idx_name will speed up queries that filter by both first_name and last_name.

WHAT ARE TRANSACTIONS IN SQL?

A **transaction** is a sequence of one or more SQL operations that are executed as a single unit. Transactions ensure the integrity of a database by ensuring that either all operations in a transaction are committed (saved) or none are, in the event of a failure.

ACID Properties of Transactions

Transactions are governed by the **ACID properties**:

1. **Atomicity**: Ensures that a transaction is treated as a single unit, which either succeeds or fails as a whole.
2. **Consistency**: Ensures that the database remains in a consistent state before and after the transaction.
3. **Isolation**: Ensures that the operations of one transaction are isolated from those of other concurrent transactions.
4. **Durability**: Ensures that the changes made by a committed transaction are permanent, even in the event of a system crash.

Transaction Commands in SQL

1. **BEGIN TRANSACTION**: Starts a new transaction.
2. **COMMIT**: Saves the changes made during the transaction.
3. **ROLLBACK**: Reverts the changes made during the transaction.

Example of Using Transactions

```
 BEGIN TRANSACTION;
```

```
UPDATE employees
SET salary = salary + 5000
WHERE department_id = 3;
UPDATE departments
SET budget = budget - 5000
WHERE department_id = 3;
COMMIT;
```

Output:

This series of commands starts a transaction, updates the salaries of employees in department 3, adjusts the department's budget, and commits the changes to the database.

What are Stored Procedures in SQL?

A **stored procedure** is a precompiled collection of one or more SQL statements stored in the database. Stored procedures are used to encapsulate business logic, perform operations like inserting, updating, and deleting data, and handle complex queries.

Creating a Stored Procedure

Here’s an example of creating a stored procedure to calculate the bonus of employees:

```
CREATE PROCEDURE calculate_bonus()
BEGIN
   UPDATE employees
   SET salary = salary + 1000
   WHERE performance_rating = 'Excellent';
END;
```

Output:

This stored procedure updates the salaries of employees with an excellent performance rating by adding 1000 to their current salary.

WHAT ARE FUNCTIONS IN SQL?

A **function** in SQL is similar to a stored procedure but is designed to return a value. Functions can be used to perform operations like calculations, string manipulations, or aggregation on data.

Creating a Function

Here’s an example of creating a function that calculates the annual salary based on the monthly salary:

```
 CREATE FUNCTION calculate_annual_salary(monthly_salary DECIMAL)
RETURNS DECIMAL
BEGIN
   RETURN monthly_salary * 12;
END;
```

Output:

This function takes a monthly_salary as input and returns the annual salary by multiplying the monthly salary by 12.

Example of Using a Function

You can call the calculate_annual_salary function as part of a query to retrieve the annual salary of employees:

```
SELECT name, calculate_annual_salary(salary) AS annual_salary
FROM employees;
```

Output:

This query will return the names of employees along with their annual salaries.

TYPES OF JOINS

There are several types of joins in SQL:

1. **INNER JOIN**: Returns records that have matching values in both tables.
2. **LEFT JOIN (or LEFT OUTER JOIN)**: Returns all records from the left table, and the matched records from the right table. If there is no match, NULL values are returned for the right table's columns.
3. **RIGHT JOIN (or RIGHT OUTER JOIN)**: Similar to the LEFT JOIN, but it returns all records from the right table and the matching records from the left table.
4. **FULL JOIN (or FULL OUTER JOIN)**: Returns all records when there is a match in either left or right table. Rows without matches will have NULL values in columns from the table with no match.
5. **SELF JOIN**: Joins a table with itse

Example of INNER JOIN

Suppose you have two tables: employees and departments. You want to get a list of all employees along with the department names they belong to. This can be achieved using an INNER JOIN:

```
SELECT employees.name, employees.salary, departments.department_name
FROM employees
INNER JOIN departments
ON employees.department_id = departments.department_id;
```

Output:

This query returns a list of employee names, their salaries, and the corresponding department names, but only for employees who are assigned to a department.

Example of LEFT JOIN

If you want to include all employees, even those who don't belong to any department, you can use a LEFT JOIN:

```
SELECT employees.name, employees.salary, departments.department_name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.department_id;
```

Output:

This query will return a list of all employees, their salaries, and department names. Employees who are not assigned to a department will show NULL for the department_name.

Example of RIGHT JOIN

Similarly, a RIGHT JOIN ensures that all records from the right table are included, even if there is no matching record in the left table:

```
SELECT employees.name, employees.salary, departments.department_name
```

```
FROM employees
RIGHT JOIN departments
ON employees.department_id = departments.department_id;
```

Output:

This query returns a list of all departments and employees in those departments. Departments with no employees will show NULL for employee data.

Example of FULL JOIN

A FULL JOIN returns all rows when there is a match in one of the tables:

```
SELECT employees.name, employees.salary, departments.department_name
FROM employees
FULL OUTER JOIN departments
ON employees.department_id = departments.department_id;
```

Output:

This query returns a list of all employees and departments, including employees without departments and departments without employees.

Example of Self Join

A **Self Join** is used to join a table with itself. For instance, if you want to list managers and the employees they manage, assuming that the employees table has a manager_id column:

```
SELECT e1.name AS employee_name, e2.name AS manager_name
FROM employees e1
LEFT JOIN employees e2
```

```
ON e1.manager_id = e2.employee_id;
```

Output:

This query lists employees along with their respective managers. Employees without managers will show NULL for the manager_name.

SQL GROUPING AND AGGREGATING DATA

SQL allows you to **group** rows and apply **aggregate functions** to summarize data. This is typically done using the GROUP BY clause along with aggregate functions such as COUNT, SUM, AVG, MAX, and MIN.

Using COUNT, SUM, and AVG Functions

For example, if you want to count the number of employees in each department:

```
SELECT department_id, COUNT(*) AS number_of_employees
FROM employees
GROUP BY department_id;
```

Output:

This query returns the number of employees in each department.

Example of Using SUM and AVG

If you want to find the total salary and average salary of employees in each department:

```
SELECT department_id, SUM(salary) AS total_salary, AVG(salary) AS average_salary
```

```
FROM employees
GROUP BY department_id;
```

Output:

This query returns the total and average salary of employees in each department.

HAVING CLAUSE

The HAVING clause is used to filter the results of aggregate functions, similar to how WHERE filters individual rows. It is applied after the GROUP BY clause.

For instance, to find departments with an average salary greater than 50,000:

```
SELECT department_id, AVG(salary) AS average_salary
FROM employees
GROUP BY department_id
HAVING AVG(salary) > 50000;
```

Output:

This query returns departments with an average salary greater than 50,000.

SQL CONSTRAINTS

Constraints are rules applied to the columns of a table to ensure data integrity and consistency. There are several types of constraints in SQL.

Types of Constraints

1. **NOT NULL**: Ensures that a column cannot have a NULL value.
2. **UNIQUE**: Ensures that all values in a column are unique.
3. **PRIMARY KEY**: Uniquely identifies each record in a table and automatically applies the NOT NULL and UNIQUE constraints.
4. **FOREIGN KEY**: Ensures the integrity of data between two tables by linking a column in one table to a primary key in another.
5. **CHECK**: Ensures that the values in a column satisfy a specific condition.
6. **DEFAULT**: Specifies a default value for a column if no value is provided.

Example of NOT NULL and UNIQUE Constraints

```
CREATE TABLE employees (
   employee_id INT NOT NULL,
   name VARCHAR(100) UNIQUE,
   department_id INT NOT NULL
);
```

Output:

This query creates an employees table with employee_id and department_id as mandatory (NOT NULL), and name as unique.

Example of PRIMARY KEY and FOREIGN KEY Constraints

```
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100)
);
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(100),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

Output:

This creates a departments table with a primary key and an employees table with a foreign key constraint linking department_id to departments.department_id.

SQL OPERATORS

SQL operators allow you to perform operations on values in a query. Some commonly used SQL operators include:

1. **Comparison Operators**: =, !=, <, >, <=, >=
2. **Logical Operators**: AND, OR, NOT
3. **Wildcard Operators**: % (represents any sequence of characters), _ (represents a single character)
4. **BETWEEN**: Used to filter results within a range.
5. **IN**: Allows you to specify multiple values for a column.
6. **LIKE**: Used to search for a pattern in a column.

Example of Using Comparison Operators

To find all employees with a salary greater than 50,000:

```
SELECT * FROM employees
WHERE salary > 50000;
```

Output:

This query returns employees who have a salary greater than 50,000.

Example of Using IN Operator

If you want to select employees who belong to specific departments:

```
SELECT * FROM employees
WHERE department_id IN (1, 2, 3);
```

Output:

This query returns employees who belong to departments 1, 2, or 3.

Example of Using LIKE Operator

To find employees whose name starts with 'J':

```
SELECT * FROM employees
WHERE name LIKE 'J%';
```

Output:

This query returns employees whose name starts with the letter 'J'.

Stored Procedures

A **Stored Procedure** is a set of SQL statements that are stored in the database and executed as a single unit. They can take parameters, execute multiple SQL statements, and even return results.

CREATING A STORED PROCEDURE

Here's an example of creating a stored procedure that calculates the total salary of employees in a given department:

```
DELIMITER //

CREATE PROCEDURE CalculateTotalSalary(IN dept_id INT)
BEGIN
   SELECT SUM(salary) AS total_salary
   FROM employees
   WHERE department_id = dept_id;
END //

DELIMITER ;
```

Explanation:

- DELIMITER //: Changes the delimiter temporarily so that semicolons can be used within the procedure.
- IN dept_id INT: This defines an input parameter dept_id of type INT.
- SUM(salary) AS total_salary: This aggregates the salaries of employees in the specified department.

CALLING THE STORED PROCEDURE

To call the stored procedure and get the total salary for department 1:

```
 CALL CalculateTotalSalary(1);
```

Output:

The result will show the total salary for all employees in department 1.

Functions

A **Function** in SQL is similar to a stored procedure but returns a single value. Functions are commonly used for computations like calculating totals, averages, or transforming data.

CREATING A FUNCTION

Here's an example of creating a function that returns the average salary of employees in a specific department:

```
 DELIMITER //
CREATE FUNCTION GetAverageSalary(dept_id INT)
RETURNS DECIMAL(10,2)
BEGIN
   DECLARE avg_salary DECIMAL(10,2);
   SELECT AVG(salary) INTO avg_salary
   FROM employees
   WHERE department_id = dept_id;
   RETURN avg_salary;
END //

DELIMITER ;
```

Explanation:

- RETURNS DECIMAL(10,2): Specifies the return type of the function (in this case, a decimal number with 2 decimal places).
- SELECT AVG(salary) INTO avg_salary: Retrieves the average salary into the variable avg_salary.
- RETURN avg_salary: Returns the calculated average salary.

CALLING THE FUNCTION

To call the function and get the average salary for department 2:

```
SELECT GetAverageSalary(2);
```

Output:

The result will return the average salary of employees in department 2.

Triggers

A **Trigger** is a set of SQL statements that are automatically executed in response to certain events on a particular table or view. Triggers are useful for enforcing business rules, data validation, and maintaining data integrity.

Creating a Trigger

Here's an example of a trigger that automatically updates an employee's last modified date whenever their record is updated:

```
DELIMITER //
CREATE TRIGGER update_employee_timestamp
BEFORE UPDATE ON employees
```

```
FOR EACH ROW
BEGIN
   SET NEW.last_modified = NOW();
END //
DELIMITER ;
```

Explanation:

- BEFORE UPDATE: Specifies that the trigger will fire before an update operation.
- FOR EACH ROW: Ensures the trigger is applied to each affected row.
- SET NEW.last_modified = NOW(): Updates the last_modified column with the current date and time.

TRIGGER EXECUTION

This trigger will automatically execute whenever an employee record is updated, ensuring that the last_modified field is always up-to-date.

Views

A **View** is a virtual table based on the result of a query. Views can simplify complex queries, enhance security by restricting access to sensitive data, and provide a consistent interface to users.

Creating a View

Here's an example of creating a view that displays employee names along with their department names:

```
CREATE VIEW employee_view AS
SELECT e.name, e.salary, d.department_name
```

```
FROM employees e
JOIN departments d
ON e.department_id = d.department_id;
```

Explanation:

- CREATE VIEW employee_view AS: Defines the view employee_view.
- The query joins the employees table and departments table, providing a list of employee names and their respective department names.

Using the View

To query the view and get a list of all employees and their departments:

```
SELECT * FROM employee_view;
```

Output:

This query returns the employee names, their salaries, and department names as specified in the view.

INDEXES

Indexes are used to speed up the retrieval of data from a table. When you create an index on a column, the database creates an internal structure that allows for faster searches.

Creating an Index

Here's an example of creating an index on the department_id column in the employees table:

```
CREATE INDEX idx_department_id
ON employees(department_id);
```

Explanation:

- CREATE INDEX idx_department_id: Creates an index named idx_department_id.
- ON employees(department_id): Specifies that the index is based on the department_id column. Indexes can significantly improve the performance of queries that involve searching, sorting, or joining on indexed columns. However, they can slow down data modifications (INSERT, UPDATE, DELETE) because the index must also be updated.

SQL DATA TYPES

Understanding the available **data types** in SQL is crucial for designing efficient and robust databases. Here's an overview of common data types:

1. **INT**: Used to store integer values.
2. **VARCHAR(n)**: A variable-length string, where n specifies the maximum number of characters.
3. **TEXT**: A string of variable length, typically used for longer text.
4. **DATE**: Used to store date values in the format YYYY-MM-DD.
5. **DECIMAL(p,s)**: Used to store precise numeric values, where p is the total number of digits and s is the number of digits to the right of the decimal point.
6. **BLOB**: Used to store binary data, such as images or files.

WORKING WITH BLOB DATA TYPE

The **BLOB** data type (Binary Large Object) is used to store large amounts of binary data, such as images, files, and other media. Here's an example of inserting and retrieving a BLOB:

Inserting a BLOB

```
INSERT INTO products (product_id, product_name, product_image)
VALUES (1, 'Diamond Ring', LOAD_FILE('/path/to/image.jpg'));
```

Explanation:

- LOAD_FILE('/path/to/image.jpg'): Loads the image file from the specified path into the product_image column.

Retrieving a BLOB

```
SELECT product_image
FROM products
WHERE product_id = 1;
```

Output:

This query retrieves the binary image data for the product with product_id = 1.

1. INTRODUCTION TO HTML AND CSS:

What is HTML?

HTML stands for **Hypertext Markup Language**. It is the standard language used to create and structure content on the web. It tells the web browser how to display text, links, images, and other forms of multimedia on a webpage. HTML sets up the basic structure of a website, and then CSS and JavaScript add style and interactivity to make it look and function better.

Features of HTML:

- It is easy to learn and easy to use.
- It is platform-independent.

- Images, videos, and audio can be added to a web page.
- Hypertext can be added to the text.
- It is a markup language.

History of HTML:

1. **Birth of HTML (1989–1991)**
   - **Tim Berners-Lee**, a scientist at CERN, proposed the idea of HTML in 1989.
   - He created the first version in 1991 with 18 simple tags to structure documents.
   - The goal was to share and link documents across the internet.
2. **HTML 2.0 (1995)**
   - First official HTML standard by the Internet Engineering Task Force (IETF).
   - Included basic tags like headings, paragraphs, lists, and forms.
   - Supported images and links.
3. **HTML 3.2 (1997)**
   - Managed by the W3C (World Wide Web Consortium).
   - Introduced tables, applets, and support for scripting languages like JavaScript.
   - Made web pages more interactive.
4. **HTML 4.0 (1999)**
   - Focused on better structure and accessibility.
   - Added support for multimedia (audio and video embedding).
   - Introduced id and class for improved styling with CSS.
5. **XHTML (2000)**
   - A stricter version of HTML, based on XML.
   - Focused on clean and well-structured code.
   - Adoption was limited due to its strictness.
6. **HTML5 (2008–Present)**
   - A modern, flexible version developed by W3C and WHATWG.
   - Key features:

        - Support for multimedia (audio, video).
        - New semantic tags like <header>, <footer>, <article>.
        - APIs for offline storage, geolocation, and more.
    - Focused on responsive and mobile-friendly design.

What is CSS?

CSS (Cascading Style Sheets) is a language designed to simplify the process of making web pages presentable. It allows you to apply styles to HTML documents, describing how a webpage should look by prescribing colors, fonts, spacing, and positioning. CSS provides developers and designers with powerful control over the presentation of HTML elements

Key Features of CSS:

- Separation of Content and Style: CSS allows developers to separate the content (HTML) from its visual representation, making maintenance easier and more efficient.
- Styling Rules: CSS uses rules to apply styles to elements. Each rule consists of a selector and a declaration block.
- Cascading Nature: The "Cascading" in CSS means that styles can fall (or cascade) from one style sheet to another, and multiple style rules can be combined.
- Responsive Design: CSS provides tools for creating responsive designs that adapt to different screen sizes and devices.
- Animation and Interaction: CSS supports animations and transitions, enhancing user interaction and visual appeal.

History of CSS:

1. Before CSS (1990–1994)
    - HTML was used for both content and design, making web pages hard to manage.
    - Developers realized the need to separate design (styles) from content.
2. CSS Idea (1994)
    - Håkon Wium Lie proposed CSS to style web pages separately from HTML.

    - The concept of "Cascading" meant styles would apply in order of priority (inline > internal > external).

3. CSS1 (1996)
    - First version of CSS released by W3C.
    - Allowed basic styles like colors, fonts, margins, and text alignment.
    - Limited browser support.
4. CSS2 (1998)
    - Added features like:
        - Positioning (absolute, relative).
        - Media types (screen, print).
        - Z-index for layering.
    - Browsers didn't fully support it, causing inconsistencies.
5. CSS3 (2000–2012)
    - Divided into smaller parts (modules) for faster updates.
    - New features like:
        - Rounded corners, shadows, gradients.
        - Animations, transitions.
        - Media queries for responsive design.
    - Modern browsers like Chrome and Firefox supported it well.
6. Modern CSS (2012–Now)
    - Responsive layouts with Flexbox and Grid became popular.
    - CSS continues to evolve, adding new features like nesting and custom properties.

2.BASIC HTML DOCUMENT STRUCTURE:

HTML Structure:

```
<!DOCTYPE html>

<html>
```

```
<head>
   <title>My First Webpage</title>
</head>
<body>
   <h1>Welcome to My Webpage</h1>
  <p>This is my first paragraph of text!</p>
</body>
</html>
```

1. **<!DOCTYPE html>**: Specifies that the document is an HTML5 document, ensuring the page is rendered correctly in modern browsers.
2. **<html>**: The outermost container for all the HTML content on the page, signaling the start of the HTML structure.
3. **<head>**: Holds essential information about the document, such as the page title, character encoding, and links to stylesheets or scripts.
4. **<body>**: The section where all visible content, like text, images, and links, is placed for display in the browser.
5. **<h1>**: A top-level heading tag used to display the most important title or section heading on a page.
6. **<p>**: A tag used for wrapping paragraphs of text, making content easier to read by separating blocks of text.

CSS Syntax:

```
selector {
  property: value;
}
```

Example CSS:

```
/* Style for the entire body */

body {

font-family: Arial, sans-serif;  /* Sets the font to Arial */

line-height: 1.5;                /* Increases line spacing for readability */

margin: 0;                       /* Removes default margin */

padding: 20px;                   /* Adds padding inside the page */

background-color: #f4f4f4;       /* Sets background color */

}

/* Style for the main heading */

h1 {

font-size: 2em;                  /* Makes the main heading larger */

text-align: center;              /* Centers the main heading */

color: #333;                     /* Sets the color to a dark gray */

}

/* Style for subheadings */

h2 {

font-size: 1.5em;                /* Makes subheadings slightly smaller */

margin-top: 30px;                /* Adds space above the subheading */
color: #555;                    /* Sets a lighter gray color */}

/* Style for paragraphs */

p {

font-size: 1em;                  /* Sets the font size of paragraphs */

color: #222;                    /* Dark color for the text */

text-align: justify;            /* Justifies the text for a neat look */

margin-bottom: 15px;            /* Adds space after each paragraph */

}

/* Links style */

a {

color: #1a73e8;                  /* Sets the link color */

text-decoration: none;          /* Removes underline */

}

a:hover {

text-decoration: underline;      /* Underlines the link when hovered */

}
```

Applying CSS to HTML:

1.Inline CSS:

- **What It Is**: Apply styles directly to individual HTML elements using the style attribute.

Example:

```
<h1 style="color: blue;">Welcome to My eBook</h1>

<p style="font-size: 16px; color: gray;">This is the first chapter of my eBook.</p>
```

Explanation: The styles are added directly to the h1 and p tags using the style attribute.

2. Internal CSS:

- **What It Is**: Add styles in the <style> tag inside the <head> section of the same HTML page.

Example:

```
<!DOCTYPE html>

<html>

<head>

  <title>My eBook</title>

  <style>

    body {

      font-family: Arial, sans-serif;

      background-color: #f0f0f0;

    }

    h1 {

      color: green;
```

```
            text-align: center;
        }
        p {
            font-size: 18px;
            color: #333;
        }
    </style>
</head>
<body>
    <h1>Welcome to My eBook</h1>
    <p>This is the first chapter of my eBook.</p>
</body>
</html>
```

Explanation: The styles are placed in the <style> section at the top of the HTML document and apply to all content in the body.

External CSS:

- **What It Is**: Use a separate CSS file to style the HTML document. This method is the best for large eBooks with multiple pages

Steps**:**

Create a CSS file (e.g., ebook-style.css):

```
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
}
h1 {
    color: purple;
    text-align: center;
}
p {
    font-size: 18px;
    color: #333;
}
```

- **Link the CSS file to the HTML page**:

```
<!DOCTYPE html>
<html>
<head>
    <title>My eBook</title>
    <link rel="stylesheet" href="ebook-style.css">
</head>
<body>
    <h1>Welcome to My eBook</h1>
    <p>This is the first chapter of my eBook.</p>
</body>
</html>
```

Explanation: The <link> tag in the <head> section links the external ebook-style.css file to the HTML page, applying the styles defined in that file

3.HTML ELEMENTS:

1. **<html>**

- **What It Does**: This is the root element that wraps all your HTML content. Every HTML document starts with this tag.
- **Example**:

```
<html>
   <!-- All content goes here -->
</html>
```

2. **<head>**

- **What It Does**: Contains metadata about your eBook, such as the title, links to stylesheets, and scripts.
- **Example**:

```
<head>
   <title>My eBook</title>
</head>
```

3. **<title>**

- **What It Does**: Sets the title that appears on the browser tab when the eBook is opened.
- **Example**:

```
<title>Chapter 1: Introduction</title>
```

4. **<body>**

- **What It Does**: Contains the visible content of the eBook that readers see, such as text, images, and links.
- **Example**:

```
<body>
   <h1>Welcome to My eBook</h1>
   <p>This is the first paragraph.</p>
</body>
```

5. **<h1>**, **<h2>**, **<h3>** (Headings)

- **What It Does**: Headings define titles and subheadings. <h1> is the main title, <h2> is for section titles, and so on.
- **Example**:

```
<h1>Welcome to My eBook</h1>
<h2>Chapter 1: Introduction</h2>
```

6. **<p>** (Paragraph)

- **What It Does**: Used for regular text. Each <p> tag contains a paragraph of text.
- **Example**:

```
<p>This is a paragraph of text in the eBook.</p>
```

7. <img> (Image)

- **What It Does**: Adds an image to your eBook. You need to specify the image's source using the src attribute.
- **Example**:

```
<img src="cover-image.jpg" alt="Cover Image of My eBook">
```

8. **<a>** (Link)

- **What It Does**: Creates clickable links. You use the href attribute to define the destination URL.
- **Example**:

```
<a href="https://www.example.com">Visit Example Website</a>
```

9. **<ul>** (Unordered List)

- **What It Does**: Creates a list of items without numbers. Each item inside the list is defined by <li>.
- **Example**:

```
<ul>
   <li>Introduction</li>
   <li>Chapter 1: Basics</li>
   <li>Chapter 2: Advanced Topics</li>
</ul>
```

10. **<ol>** (Ordered List)

- **What It Does**: Creates a list of items with numbers or letters. Each item is defined by <li>.
- **Example**:

```
<ol>
   <li>Step 1: Open the eBook</li>
   <li>Step 2: Read the first chapter</li>
   <li>Step 3: Continue to the next chapter</li>
</ol>
```

11. **<strong>**

- **What It Does**: Makes text bold and important, usually to emphasize a word or phrase.
- **Example**:

```
<strong>This is an important note!</strong>
```

12. **<em>**

- **What It Does**: Makes text italic, usually used to emphasize a word or phrase.
- **Example**:

```
<em>Read this carefully.</em>
```

13. **<blockquote>**

- **What It Does**: Displays a block of quoted text.
- **Example**:

```
<blockquote>
  "This is a quote from another book."
</blockquote>
```

14. **<br>** (Line Break)

- **What It Does**: Inserts a line break (new line) in the content.
- **Example**:

```
This is the first line.<br>
This is the second line.
```

15. **<hr>** (Horizontal Rule)

- **What It Does**: Creates a horizontal line to separate sections of the content.
- **Example**:

```
<hr>
```

4.HTML STYLES:

1. Inline CSS for Styling:

- **What It Is**: Inline CSS is when you apply styles directly to a specific HTML element using the style attribute.
- **Usage**: This method is best for quick and small changes.
- **Example**:

```
<h1 style="color: blue; text-align: center;">Welcome to My eBook</h1>
<p style="font-size: 18px; color: gray;">This is the first chapter of my eBook.</p>
```

  - **Explanation**: Here, the style attribute is used to change the color and text alignment of the heading (h1), and the font size and color of the paragraph (p).

2. Internal CSS for Styling:

- **What It Is**: Internal CSS is used when you define styles in the <style> tag within the <head> section of your HTML page.
- **Usage**: This is good for styling a single page of the eBook.
- **Example**:

```
<!DOCTYPE html>
```

```
<html>
<head>
  <title>My eBook</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      background-color: #f4f4f4;
    }
    h1 {
      color: darkblue;
      text-align: center;
    }
    p {
      font-size: 16px;
      color: #333;
    }
  </style>
</head>
<body>
  <h1>Chapter 1: Introduction</h1>
  <p>Welcome to the first chapter of my eBook!</p>
</body>
</html>
```

- **Explanation**: The <style> tag in the <head> section contains CSS rules for the body, h1, and p elements. This style will apply to the entire page.

3. External CSS for Styling:

- **What It Is**: External CSS is the best method for large eBooks. It allows you to store styles in a separate CSS file and link that file to your HTML document.

- **Usage**: Perfect for large eBooks with many chapters, ensuring consistent design across all pages.

Steps:

1. **Create a CSS File** (e.g., ebook-style.css):

```
body {
   font-family: "Times New Roman", serif;
   line-height: 1.8;
   background-color: #fff;
}
h1 {
   color: purple;
   text-align: center;
}
p {
   font-size: 18px;
   color: #555;
}
```

2. **Link the CSS File to Your HTML Page**:

```
<!DOCTYPE html>
<html>
<head>
   <title>My eBook</title>
   <link rel="stylesheet" type="text/css" href="ebook-style.css">
</head>
<body>
   <h1>Welcome to My eBook</h1>
```

```
    <p>This is the first chapter styled using external CSS.</p>
</body>
</html>
```

  - **Explanation**: The <link> tag links the ebook-style.css file to the HTML document, applying the styles defined in that file to the page.

5. HTML HEADING:

Basic Syntax of HTML Headings:

There are six levels of headings in HTML, from <h1> to <h6>. The <h1> tag is the most important (largest), while the <h6> tag is the least important (smallest).

Example:

```
<h1>Main Title of the eBook</h1>
<h2>Chapter 1: Introduction</h2>
<h3>Section 1.1: What is HTML?</h3>
<h4>Subsection 1.1.1: HTML Syntax</h4>
<h5>Details about HTML Syntax</h5>
<h6>Additional Information</h6>
```

- **Explanation**: The text inside each heading tag is displayed with different sizes and importance. <h1> is used for the main title, <h2> for chapters, and so on for sections and subsections.

6. HTML FORMATTING:

1. Text Formatting Tags

HTML offers several tags to format text. These tags help you **bold**, **italicize**, **underline**, and **strike-through** text.

Bold Text:

- The <b> or <strong> tags are used to make text bold.

```
<p><b>This is bold text.</b></p>
<p><strong>This is also bold text.</strong></p>
```

- **Explanation**: The <b> tag makes text bold for styling purposes, while <strong> has a semantic meaning of importance, making it bold.

Italic Text:

- The <i> or <em> tags are used to italicize text.

```
<p><i>This is italic text.</i></p>
<p><em>This is also italicized text.</em></p>
```

- **Explanation**: The <i> tag is for text that is styled as italic, and <em> is used for emphasis, which also italicizes the text.

Underlined Text:

- The <u> tag is used to underline text.

```
<p><u>This is underlined text.</u></p>
```

- **Explanation**: The <u> tag underlines the text for emphasis.

Strike-through Text:

- The <s> or <del> tags are used to create strike-through text.

```
<p><s>This text has a strike-through.</s></p>
<p><del>This text is also struck through.</del></p>
```

- **Explanation**: The <s> and <del> tags are used for indicating deleted or outdated text.

2. Paragraph and Line Breaks

In HTML, paragraphs and line breaks are essential for organizing text into readable chunks.

Paragraph:

- The <p> tag is used to create a paragraph of text.

```
<p>This is a paragraph of text in the eBook.</p>
```

- **Explanation**: The <p> tag automatically separates paragraphs with space, making the content easier to read.

Line Break:

- The <br> tag is used to break the line without creating a new paragraph.

```
<p>This is a paragraph.<br>This line breaks to a new line.</p>
```

- **Explanation**: The <br> tag forces the next content to appear on a new line without starting a new paragraph.

3. Lists for Organizing Content

HTML allows you to create both ordered and unordered lists, making your content structured and easy to follow.

Unordered List (Bulleted List):

- Use the <ul> (unordered list) and <li> (list item) tags to create a bulleted list.

```
<ul>
  <li>Introduction to HTML</li>
```

```
    <li>HTML Structure</li>
    <li>HTML Tags and Attributes</li>
</ul>
```

- **Explanation**: The <ul> tag creates a bulleted list, and each <li> tag defines a list item.

Ordered List (Numbered List):

- Use the <ol> (ordered list) and <li> tags to create a numbered list.

```
<ol>
    <li>Chapter 1: Introduction</li>
    <li>Chapter 2: Basics of HTML</li>
    <li>Chapter 3: Advanced HTML Techniques</li>
</ol>
```

- **Explanation**: The <ol> tag creates a numbered list, and each <li> tag defines a list item.

4. Text Alignment

HTML allows you to align text to the left, right, center, or justify it.

Left Align Text:

By default, text is aligned to the left, but you can specify it using the align attribute or CSS.

```
<p align="left">This text is aligned to the left.</p>
```

Center Align Text:

```
<p align="center">This text is centered.</p>
```

Right Align Text:

<p align="right">This text is aligned to the right.</p>

Justify Text:

<p style="text-align: justify;">This text is justified, meaning it stretches to fit both the left and right margins.</p>

5. Font Formatting

You can change the font style, size, and color of the text to make it more attractive.

Changing Font Size:

<p style="font-size: 18px;">This is a paragraph with a larger font size.</p>

Changing Font Color:

<p style="color: red;">This text is red in color.</p>

Changing Font Family:

<p style="font-family: Arial, sans-serif;">This text uses the Arial font family.</p>

6. Using Blockquotes for Quotes

- The <blockquote> tag is used to display long quotes or cited text, often indented.

```
<blockquote>
   <p>"HTML is the foundation of web development." - Web Development Guru</p>
</blockquote>
```

- **Explanation**: The <blockquote> tag is used for quoting or citing text from other sources.

7. Combining Formatting with Other Tags

You can combine different formatting tags within your eBook content. For example, you can bold a word in an italicized sentence or underline an entire paragraph.

Example:

```
<p><b><i>This text is both bold and italicized.</i></b></p>
```

- **Explanation**: Here, both the <b> and <i> tags are used to apply bold and italic styles to the text.

7. HTML LINKS:

1.Basic Syntax of an HTML Link:

The basic syntax for creating a link is:

```
<a href="URL">Link Text</a>
```

- **href**: Specifies the URL (web address) of the page or document you want to link to.
- **Link Text**: The clickable text that appears to the reader.

Example:

```
<a href="https://example.com">Visit Example Website</a>
```

- **Explanation**: This link will take the reader to the "Example Website" when clicked.

2. Types of Links

HTML links can point to different types of resources:

a. Linking to an External Website

```
<a href="https://www.wikipedia.org">Visit Wikipedia</a>
```

- **Explanation**: This link takes the reader to an external website (Wikipedia).

b. Linking to a Section in the Same Page

Use an **ID** to link to a specific section within the same page.

```
<a href="#chapter1">Go to Chapter 1</a>

<h2 id="chapter1">Chapter 1: Introduction</h2>
```

- **Explanation**: Clicking "Go to Chapter 1" will scroll the reader to the section with the ID chapter1.

c. Linking to Another Page in the Same Website

```
<a href="about.html">About This eBook</a>
```

- **Explanation**: This link opens the "about.html" page located in the same website or project.

d. Linking to a File for Download

```
<a href="ebook.pdf" download>Download eBook</a>
```

- **Explanation**: Clicking this link will download the ebook.pdf file.

3. Open Link in a New Tab

To open a link in a new tab, use the target="_blank" attribute.

```
<a href="https://example.com" target="_blank">Visit Example in New Tab</a>
```

- **Explanation**: This ensures the reader stays on the current eBook page while opening the link in a new tab.

4. Adding a Tooltip

You can add a tooltip to your link using the title attribute. The tooltip appears when the reader hovers over the link.

<a href="https://example.com" title="Click to visit Example Website">Visit Example</a>

- **Explanation**: When the reader hovers over the link, the tooltip "Click to visit Example Website" appears.

5. Styling Links

You can style links using **CSS** to change their color, font, or hover effects.

Example:

<a href="https://example.com" style="color: blue; text-decoration: none;">Styled Link</a>

- **Explanation**: The link text will appear in blue and without an underline.

Example of Hover Effect:

<a href="https://example.com" style="color: blue;" onmouseover="this.style.color='red'" onmouseout="this.style.color='blue'">Hover Over Me</a>

- **Explanation**: The link text changes to red when the reader hovers over it and back to blue when the hover ends.

6. Linking an Email Address

To create a link that opens an email client, use mailto: in the href attribute.

```
<a href="mailto:example@example.com">Email Us</a>
```

- **Explanation**: Clicking the link will open the default email client with the address example@example.com ready to send an email.

7. Linking to Phone Numbers

To create a clickable phone number link, use tel: in the href attribute.

```
<a href="tel:+1234567890">Call Us</a>
```

- **Explanation**: Clicking the link will prompt a call to the number +1234567890 on mobile devices.

8. Disabled Links

To create a link that isn't clickable, omit the href attribute or set it to #.

```
<a href="#">This link is disabled.</a>
```

- **Explanation**: This can be used as a placeholder for future links.

8. HTML TABLES:

1. Basic Structure of an HTML Table

An HTML table is created using the <table> tag, with rows defined by <tr> (table row) and cells by <td> (table data).

Example:

```
<table border="1">
```

```
    <tr>
        <td>Row 1, Cell 1</td>
        <td>Row 1, Cell 2</td>
    </tr>
    <tr>
        <td>Row 2, Cell 1</td>
        <td>Row 2, Cell 2</td>
    </tr>
</table>
```

- **Explanation**: This table has two rows, each containing two cells. The border="1" adds a border around the table.

2. Adding a Table Header

You can use the <th> tag to create header cells, which are bold and centered by default.

Example:

```
<table border="1">
    <tr>
        <th>Header 1</th>
        <th>Header 2</th>
    </tr>
    <tr>
        <td>Data 1</td>
        <td>Data 2</td>
    </tr>
</table>
```

- **Explanation**: The <th> tag is used for header cells, making the first row a header for the table.

3. Adding a Caption to the Table

The <caption> tag is used to add a title or description to the table.

Example:

```
<table border="1">
    <caption>Sample Data Table</caption>
    <tr>
        <th>Header 1</th>
        <th>Header 2</th>
    </tr>
    <tr>
        <td>Data 1</td>
        <td>Data 2</td>
    </tr>
</table>
```

- **Explanation**: The caption appears above the table and gives it a title.

4. Combining Rows or Columns

You can merge cells using the rowspan and colspan attributes.

Merge Columns with **colspan**:

```
<table border="1">
    <tr>
        <th colspan="2">Merged Header</th>
    </tr>
    <tr>
        <td>Data 1</td>
        <td>Data 2</td>
    </tr>
</table>
```

- **Explanation**: The colspan="2" merges two columns into one.

Merge Rows with **rowspan**:

```
<table border="1">
    <tr>
        <td rowspan="2">Merged Row</td>
        <td>Data 1</td>
    </tr>
    <tr>
        <td>Data 2</td>
    </tr>
</table>
```

- **Explanation**: The rowspan="2" merges two rows into one.

5. Adding Style to Tables

You can use CSS to style tables and make them look more attractive.

Example:

```
<table border="1" style="border-collapse: collapse; width: 50%;">
    <caption>Styled Table</caption>
    <tr style="background-color: lightgray;">
        <th>Header 1</th>
        <th>Header 2</th>
    </tr>
    <tr>
        <td style="text-align: center;">Data 1</td>
        <td>Data 2</td>
    </tr>
</table>
```

- **Explanation**:
    - The border-collapse: collapse; removes double borders.
    - Background colors and text alignment are added for better presentation.

6. Example Table

```
<table border="1" style="width: 100%; text-align: left;">
    <caption>Book Information</caption>
    <tr>
        <th>Title</th>
        <th>Author</th>
        <th>Year</th>
    </tr>
    <tr>
        <td>Learning HTML</td>
        <td>John Doe</td>
        <td>2024</td>
    </tr>
    <tr>
        <td>Web Development Basics</td>
        <td>Jane Smith</td>
        <td>2023</td>
    </tr>
</table>
```

- **Explanation**: This table organizes book information into rows and columns for easy readability.

9. HTML INPUT ELEMENTS:

1. Basic Syntax of an Input Element

The <input> tag is a self-closing tag and requires the type attribute to specify the kind of input.

<input type="text" />

- **Explanation**: This creates a simple text box for user input.

2. Common Types of Input Elements

a. Text Input

<input type="text" placeholder="Enter your name" />

- **Explanation**: Creates a single-line text field with a placeholder to guide users.

b. Password Input

<input type="password" placeholder="Enter your password" />

- **Explanation**: Creates a password field where typed characters are hidden.

c. Email Input

<input type="email" placeholder="Enter your email" />

- **Explanation**: Creates a field for email input with basic validation.

d. Number Input

<input type="number" placeholder="Enter a number" />

- **Explanation**: Creates a field for numeric input with controls for increasing or decreasing the value.

e. Radio Button

<input type="radio" name="choice" value="Option 1" /> Option 1
<input type="radio" name="choice" value="Option 2" /> Option 2

- **Explanation**: Allows users to select one option from a group of choices.

f. Checkbox

<input type="checkbox" name="subscribe" /> Subscribe to newsletter

- **Explanation**: Allows users to select or deselect an option.

g. Submit Button

<input type="submit" value="Submit" />

- **Explanation**: A button that submits form data.

h. Reset Button

<input type="reset" value="Reset" />

- **Explanation**: Clears all input fields in the form.

i. File Upload

<input type="file" />

- **Explanation**: Allows users to upload files.

j. Date Picker

```
<input type="date" />
```

- **Explanation**: Allows users to select a date.

3. Grouping Input Elements with Labels

You can use the <label> tag to associate a label with an input field, making it more user-friendly.

Example:

```
<label for="name">Name:</label>
<input type="text" id="name" placeholder="Enter your name" />
```

- **Explanation**: The for attribute in the <label> matches the id of the input field.

4. Adding a Form for Input Elements

Input elements are typically part of a form, which collects and submits data.

Example:

```
<form action="/submit" method="post">
   <label for="email">Email:</label>
   <input type="email" id="email" placeholder="Enter your email" />
   <br /><br />
   <input type="submit" value="Submit" />
</form>
```

- **Explanation**:
  - The <form> tag groups input elements.
  - The action attribute specifies where the form data is sent.

- The method attribute defines how the data is sent (post or get).

5. Placeholder Text

You can guide users by adding a placeholder attribute to input fields.

Example:

```
<input type="text" placeholder="Enter your full name" />
```

- **Explanation**: The placeholder text disappears when the user starts typing.

6. Input Validation

You can add attributes to ensure users enter data in the correct format.

Example:

```
<input type="email" placeholder="Enter your email" required />
```

- **Explanation**: The required attribute makes the field mandatory.

7. Disabled and Read-Only Inputs

Disabled Input:

```
<input type="text" value="Disabled Field" disabled />
```

- **Explanation**: A disabled input field cannot be edited.

Read-Only Input:

```
<input type="text" value="Read-Only Field" readonly />
```

- **Explanation**: A read-only field displays data but cannot be edited.

8. Styling Input Elements

You can use CSS to make input fields visually appealing.

Example:

```
<input type="text" style="border: 2px solid blue; padding: 10px; width: 200px;"
placeholder="Styled input" />
```

- **Explanation**: This input field has a blue border, padding, and custom width.

10. HTML IMAGES:

1. Basic Syntax of an HTML Image

The <img> tag is self-closing and requires the src (source) attribute to specify the image's location.

```
<img src="image.jpg" alt="Description of the image" />
```

- **src**: Specifies the file path or URL of the image.
- **alt**: Provides alternative text for the image, displayed if the image cannot load or for accessibility.

Example:

```
<img src="flower.jpg" alt="A beautiful flower" />
```

- **Explanation**: Displays an image of a flower. If the image cannot load, the text "A beautiful flower" will appear.

2. Adding a Title to an Image

You can add a title attribute to provide additional information about the image when the user hovers over it.

Example:

```
<img src="book.jpg" alt="An open book" title="This is an open book" />
```

- **Explanation**: When the user hovers over the image, the title text "This is an open book" appears.

3. Resizing an Image

You can control the size of the image using the width and height attributes.

Example:

```
<img src="landscape.jpg" alt="Landscape" width="300" height="200" />
```

- **Explanation**: The image will display at 300 pixels wide and 200 pixels tall.

4. Responsive Images

To make images adjust automatically based on screen size, use the style attribute or CSS.

Example:

```
<img src="responsive.jpg" alt="Responsive Image" style="max-width: 100%; height: auto;" />
```

- **Explanation**: The image will resize to fit the screen without distortion.

5. Linking an Image

You can make an image clickable by wrapping it in an <a> tag.

Example:

```
<a href="https://example.com">
  <img src="logo.jpg" alt="Website Logo" />
</a>
```

- **Explanation**: Clicking on the image takes the user to "https://example.com."

6. Using Images from External URLs

You can use images hosted on other websites by providing the full URL in the src attribute.

Example:

```
<img src="https://www.example.com/image.jpg" alt="Example Image" />
```

- **Explanation**: Displays an image from an external source.

7. Adding a Placeholder Image

If you don’t have an image ready, you can use a placeholder image.

Example:

```
<img src="https://via.placeholder.com/150" alt="Placeholder Image" />
```

- **Explanation**: Displays a placeholder image with a specified size (150x150 pixels).

11. CSS SELECTORS:

1. Common CSS Selectors

a. Element Selector

Targets all elements of a specific type.

```
p {
   color: blue;
}
```

- **Example**: Styles all <p> elements with blue text.

b. Class Selector

Targets elements with a specific class attribute.

```
.classname {
   color: green;
}
```

- **Example**: Styles all elements with the class "classname" to have green text.

c. ID Selector

Targets an element with a specific id attribute.

```
#idname {
   color: red;
}
```

- **Example**: Styles the element with the ID "idname" to have red text.

d. Universal Selector

Targets all elements in the document.

```
* {
    color: black;
}
```

- **Example**: Sets the text color of all elements to black.

e. Attribute Selector

Targets elements based on their attributes or attribute values.

```
 [attribute] {
    color: purple;
}
[attribute="value"] {
    color: orange;
}
```

- **Example**:
    - Styles all elements with a specified attribute to have purple text.
    - Styles elements with a specific attribute value to have orange text.

2. Combining Selectors

CSS selectors can be combined to style elements more precisely.

a. Descendant Selector

Targets elements nested inside another element.

```
div p {
    color: blue;
}
```

- **Example**: Styles all <p> elements inside a <div> with blue text.

b. Child Selector

Targets only the direct children of an element.

```
div > p {
   color: green;
}
```

- **Example**: Styles only the direct <p> children of a <div> with green text.

c. Adjacent Sibling Selector

Targets an element immediately following another.

```
h1 + p {
   color: red;
}
```

- **Example**: Styles the first <p> element that comes immediately after an <h1> with red text.

d. General Sibling Selector

Targets all elements preceded by a specified sibling.

```
h1 ~ p {
   color: yellow;
}
```

- **Example**: Styles all <p> elements that come after an <h1> with yellow text.

12. CSS PROPERTIES:

1. Text Styling Properties

a. color

Sets the text color.

```
p {
    color: blue;
}
```

- **Example**: Makes the text inside <p> tags blue.

b. **font-size**

Specifies the size of the text.

```
h1 {
    font-size: 24px;
}
```

- **Example**: Sets the text size of <h1> to 24 pixels.

c. **font-family**

Specifies the font style for the text.

```
body {
    font-family: Arial, sans-serif;
}
```

- **Example**: Applies the Arial font to the text.

d. **text-align**

Controls the alignment of text.

```
h2 {
   text-align: center;
}
```

- **Example**: Centers the text in <h2> elements.

e. **font-weight**

Sets the thickness of the text.

```
p {
   font-weight: bold;
}
```

- **Example**: Makes the text bold.

2. Background and Color Properties

a. **background-color**

Sets the background color of an element.

```
div {
   background-color: lightblue;
}
```

- **Example**: Adds a light blue background to a <div>.

b. **background-image**

Adds an image as a background.

```
body {
    background-image: url('background.jpg');
}
```

- **Example**: Sets an image as the background.

c. **opacity**

Adjusts the transparency of an element.

```
img {
    opacity: 0.5;
}
```

- **Example**: Makes the image 50% transparent.

3. Box Model Properties

a. **width** and **height**

Sets the dimensions of an element.

```
div {
    width: 200px;
    height: 100px;
}
```

- **Example**: Creates a box 200px wide and 100px tall.

b. **padding**

Adds space inside an element, around its content.

```
p {
    padding: 10px;
}
```

- **Example**: Adds 10px of space inside the <p> element.

c. **margin**

Adds space outside an element.

```
h1 {
    margin: 20px;
}
```

- **Example**: Adds 20px of space around the <h1> element.

d. **border**

Adds a border around an element.

```
div {
    border: 2px solid black;
}
```

- **Example**: Adds a 2px black solid border to a <div>.

4. Display and Positioning

a. **display**

Specifies how an element is displayed.

```
span {
    display: block;
}
```

- **Example**: Makes the <span> behave like a block element.

b. **position**

Defines how an element is positioned.

```
div {
   position: absolute;
   top: 50px;
   left: 100px;
}
```

- **Example**: Places the <div> 50px from the top and 100px from the left.

c. **z-index**

Controls the stack order of elements.

```
img {
   z-index: 10;
}
```

- **Example**: Brings the image to the front if elements overlap.

5. List Properties

a. **list-style-type**

Changes the bullet style of a list.

```
ul {
   list-style-type: square;
}
```

- **Example**: Changes list bullets to squares.

b. **list-style-image**

Uses an image as a bullet.

```
ul {
    list-style-image: url('bullet.png');
}
```

- **Example**: Adds a custom image as the list bullet.

6. Transition and Animation

a. **transition**

Adds smooth changes between states.

```
button {
    transition: background-color 0.3s ease;
}
```

- **Example**: Smoothly changes the button's background color.

b. **animation**

Adds animations to elements.

```
@keyframes example {
    from { background-color: red; }
    to { background-color: yellow; }
}
```

```
div {
    animation: example 2s infinite;
}
```

- **Example**: Animates a <div> to change its background color from red to yellow.

13. CSS UNITS:

1. Absolute Units

Absolute units have a fixed size and do not change based on other elements.

- **px (pixels)**: Specifies a fixed size.

  Example: font-size: 16px;
- **cm (centimeters), mm (millimeters), in (inches)**: Used for physical measurements, often in print.

  Example: width: 5cm;

2. Relative Units

Relative units adjust based on the size of their parent or other elements, making them flexible and responsive.

- **% (percent)**: Relative to the size of the parent element.

  Example: width: 50%;
- **em**: Relative to the font size of the element's parent.

  Example: font-size: 1.5em;

  (If the parent font size is 16px, this would equal 24px.)
- **rem**: Relative to the font size of the root element (<html>).

  Example: font-size: 2rem;

  (If the root font size is 16px, this would equal 32px.)

- **vw (viewport width)**: Relative to the width of the viewport (browser window).
  Example: width: 50vw;
  (50% of the viewport width.)
- **vh (viewport height)**: Relative to the height of the viewport.
  Example: height: 100vh;
  (Takes up the full height of the viewport.)

Example:

```
p {
    font-size: 1.5em;
    margin-top: 10px;
    width: 50%;
}
```

14. RESPONSIVE DESIGN:

Responsive Design: Simplified Explanation

Responsive design ensures that web pages adapt and look great on all devices, including desktops, tablets, and mobile phones. CSS provides tools like **media queries** to create responsive layouts.

Media Queries

Media queries apply specific styles based on device characteristics such as screen width, height, or orientation. This allows you to customize your page for different devices.

Example: Media Queries in Action

/* Default styles for all devices */

```
body {
   background-color: white;
   font-size: 16px;
}

/* Styles for devices with screens larger than 600px */
@media (min-width: 600px) {
   body {
      background-color: lightblue;
      font-size: 18px;
   }
}
/* Styles for devices with screens smaller than 600px */
@media (max-width: 600px) {
   body {
      background-color: lightgreen;
      font-size: 14px;
   }
}
```

How It Works:

1. **Default Styles**: The base styles apply to all devices, unless overridden by a media query.
   - Background color: white.
   - Font size: 16px.
2. **For Large Screens**:
   - Devices with a screen width **greater than 600px** have a light blue background and font size of 18px.
3. **For Small Screens**:

    - Devices with a screen width **less than or equal to 600px** have a light green background and font size of 14px.

15. CSS BORDER PROPERTIES:

1. Border Property

The border property is a shorthand for setting the width, style, and color of all four sides of an element's border.

Syntax:

```
selector {
  border: width style color;
}
```

Example:

```
div {
  border: 2px solid black;
}
```

- Adds a 2px solid black border around a <div> element.

2. Border Width

The border-width property sets the thickness of the border. It accepts specific values (e.g., pixels) or keywords like thin, medium, or thick.

Syntax:

```
selector {
  border-width: value;
}
```

Example:

```
p {
    border-width: 4px;
}
```

- Sets a 4px thick border around a <p> element.

3. Border Style

The border-style property defines the appearance of the border. Common values include:

- none: No border.
- solid: A continuous line.
- dotted: A series of dots.
- dashed: A series of dashes.
- double: Two solid lines.
- groove: A 3D grooved border.
- ridge: A 3D ridged border.
- inset: A 3D inset border.
- outset: A 3D outset border.

Syntax:

```
selector {
    border-style: value;
}
```

Example:

```
h1 {
    border-style: dashed;
}
```

- Adds a dashed border around an <h1> element.

4. Border Color

The border-color property specifies the color of the border. You can use color names, HEX codes, RGB, or RGBA values.

Syntax:

```
selector {
  border-color: color;
}
```

Example:

```
h2 {
  border-color: blue;
}
```

- Adds a blue border around an <h2> element.

16. INPUT ATTRIBUTES:

1. **placeholder**

Provides a hint or example text inside the input field to guide users on what to enter.

Example:

```
<input type="text" placeholder="Enter your name">
```

- Displays "Enter your name" inside the input field until the user types something.

2. **required**

Marks the input field as mandatory. The form will not submit unless this field is filled.

Example:

```
<input type="email" required>
```

- Ensures the user enters a valid email address before submitting the form.

3. **readonly**

Makes the input field non-editable. Users can see the value but cannot modify it.

Example:

```
<input type="text" value="Read-Only" readonly>
```

- Displays "Read-Only" in the input field, but it cannot be edited.

4. **disabled**

Disables the input field, preventing users from interacting with it.

Example:

```
<input type="text" value="Disabled" disabled>
```

- The input field appears grayed out and cannot be clicked or edited.

5. **maxlength**

Sets the maximum number of characters allowed in the input field.

Example:

```
<input type="text" maxlength="10">
```

- Limits the input to a maximum of 10 characters.

6. **size**

Specifies the width of the input field in terms of the number of characters it can display.

Example:

```
<input type="text" size="20">
```

- Creates an input field wide enough to fit approximately 20 characters.

7. **pattern**

Defines a regular expression that the input value must match. Useful for validation.

Example:

```
<input type="text" pattern="[A-Za-z]{3}">
```

- Ensures the user enters exactly three alphabetic characters.

8. **autofocus**

Automatically focuses on the input field when the web page loads.

Example:

```
<input type="text" autofocus>
```

- The input field becomes active as soon as the page is loaded.

17. CREATING A REGISTRATION FORM:

A registration form is an essential component of web development, enabling users to sign up for services, accounts, or memberships. A good form is designed to efficiently collect user information while ensuring a smooth user experience.

Basic Structure of a Registration Form

A registration form generally includes the following elements:

- Fields for **personal information** (e.g., first and last name).
- Fields for **contact details** (e.g., email).
- **Security fields** like passwords and their confirmation.
- **Checkboxes** for agreements like terms and conditions.
- **Submit button** for sending the data.

Example Registration Form:

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Registration Form</title>

<style>

   body {

      font-family: Arial, sans-serif;

      background-color: #f2f2f2;

      display: flex;
```

```
    justify-content: center;
    align-items: center;
    height: 100vh;
    margin: 0;
}
.container {
    background-color: white;
    padding: 20px;
    border-radius: 10px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    width: 300px;
}
.container h2 {
    text-align: center;
    margin-bottom: 20px;
}
.container label {
    display: block;
    margin-bottom: 5px;
}
.container input[type="text"],
.container input[type="email"],
.container input[type="password"] {
    width: 100%;
    padding: 10px;
    margin-bottom: 10px;
    border: 1px solid #ccc;
    border-radius: 5px;
}
.container input[type="submit"] {
    width: 100%;
    padding: 10px;
    background-color: #4CAF50;
    color: white;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}
.container input[type="submit"]:hover {
    background-color: #45a049;
}
.terms {
    display: flex;
    align-items: center;
```

```
        margin-bottom: 10px;
    }
    .terms input[type="checkbox"] {
        margin-right: 5px;
    }
</style>
</head>
<body>
<div class="container">
<h2>Register</h2>
<form action="register.php" method="post">
    <label for="fname">First Name</label>
    <input type="text" id="fname" name="firstname" required>
    <label for="lname">Last Name</label>
    <input type="text" id="lname" name="lastname" required>
    <label for="email">Email</label>
    <input type="email" id="email" name="email" required>
    <label for="password">Password</label>
    <input type="password" id="password" name="password" required>
    <label for="confirm_password">Confirm Password</label>
    <input type="password" id="confirm_password" name="confirm_password" required>
    <div class="terms">
```

```
        <input type="checkbox" id="terms" name="terms" required>
        <label for="terms">I agree to the <a href="#">terms and conditions</a></label>
    </div>
    <input type="submit" value="Register">
</form>
</div>
</body>
</html>
```

JAVASCRIPT

1. INTRODUCTION TO JAVASCRIPT:

What is JavaScript?

JavaScript is a **programming language used for creating dynamic content on websites**. It is a **lightweight**, **cross-platform,** and **single-threaded** programming language. JavaScript is an **interpreted** language that executes code line by line providing more flexibility.

Brief History of JavaScript:

- **1995: JavaScript was created by Brendan Eich at Netscape Communications. Amazingly, it was developed in just 10 days. Initially named Mocha, it was later renamed LiveScript and finally branded as JavaScript to ride the wave of Java's popularity, though the two languages are not related.**
- **1996: Microsoft launched JScript, a reverse-engineered version of JavaScript, to work with its Internet Explorer browser, setting the stage for competition between browsers.**
- **1997: To ensure JavaScript could work consistently across all platforms, it was standardized by ECMA International as ECMAScript. The first edition of the**

ECMAScript specification, called ECMA-262, was released, establishing a foundation for JavaScript's future.

- **2009: The introduction of ECMAScript 5 (ES5) marked a turning point. It added critical features like strict mode for safer coding, native JSON support, and new array methods to simplify development, making JavaScript more reliable and developer-friendly.**
- **2015: The release of ECMAScript 6 (ES6), or ECMAScript 2015, revolutionized JavaScript. It introduced modern programming concepts such as classes, modules, arrow functions, and promises for asynchronous code. The addition of let and const declarations improved variable management and code readability.**
- **Present: JavaScript continues to advance with annual updates to the ECMAScript standard. New features and enhancements are regularly introduced, ensuring JavaScript remains versatile, efficient, and indispensable for modern web development.**

2. KEY FEATURES OF JAVASCRIPT:

1.Client-Side Scripting

JavaScript is widely used for client-side scripting, allowing code to run directly in the user's web browser. It can manipulate the HTML structure and CSS styles to create dynamic and interactive user interfaces.

2.Interpreted and Just-In-Time Compiled

JavaScript is executed by the browser's JavaScript engine, either through line-by-line interpretation or Just-In-Time (JIT) compilation. This ensures fast execution and facilitates rapid testing and development without the need for a separate compilation step.

3.Versatility and Multi-Paradigm Support

JavaScript is a flexible, multi-paradigm language that supports object-oriented, functional, and imperative programming styles. Its versatility enables developers to build anything from simple websites to full-fledged applications, including mobile and server-side solutions.

4.Event-Driven and Asynchronous Programming

JavaScript excels in event-driven programming, enabling web pages to respond to user interactions such as clicks, keypresses, and form submissions. Additionally, its support for asynchronous programming (via callbacks, promises, and async/await) allows for efficient handling of tasks like API calls and animations.

5.Seamless Integration with Web Technologies

JavaScript works effortlessly with HTML and CSS, giving developers the ability to interact with and manipulate the Document Object Model (DOM). This integration is key to creating dynamic web pages and applications.

6.Universal Browser Support

JavaScript is supported by all major browsers, including Google Chrome, Firefox, Safari, Microsoft Edge, and others. Modern browsers ensure high compatibility and include advanced JavaScript engines like V8 (Chrome) and SpiderMonkey (Firefox).

7.Expanding Ecosystem

JavaScript's reach goes beyond the browser with environments like Node.js for server-side development and frameworks like React, Angular, and Vue.js for building rich, interactive web applications.

3. ADVANTAGES OF USING JAVASCRIPT:

1. Interactivity

JavaScript makes web pages dynamic and interactive, improving the overall user experience with features like animations, real-time updates, and form validation.

2. Fast Performance

JavaScript runs directly in the browser, so it works quickly without needing to communicate with a server for every action.

3. Versatile Usage

JavaScript is useful for both the front-end (what users see) and back-end (server-side) development, especially with tools like Node.js.

4. Rich Features

It enables the creation of interactive elements like sliders, drag-and-drop tools, and responsive forms, making websites more engaging.

5. Large Community and Tools

JavaScript has a huge community of developers and plenty of tools like React, Vue.js, and Angular that make coding easier and faster.

6. Works Everywhere

JavaScript works on all modern browsers and devices, ensuring that websites and apps function smoothly for everyone.

4. VARIABLES IN JAVASCRIPT:

A variable is like a container that stores data in your program. You can think of it as a box where you keep something to use later. Variables have names (labels) and values (the data you store in them).

Declaring Variables:

JavaScript provides three ways to declare variables:

1. **var** – The old way (less common now).
2. **let** – Used for variables that can change.
3. **const** – Used for variables that won't change.

Syntax for Declaring Variables:

let variableName = value;

- **variableName**: The name you give to the variable.
- **value**: The data stored in the variable.

Examples of Variables:

1. Using **let** (Variable Can Change)

```
let age = 25;
console.log(age); // Output: 25

age = 30; // Changing the value
console.log(age); // Output: 30
```

2. Using **const** (Variable Cannot Change)

```
const pi = 3.14;
console.log(pi); // Output: 3.14

// pi = 3.15; // This will cause an error because 'pi' is constant.
```

3. Using **var** (Older Way, Not Recommended)

```
var name = "John";
console.log(name); // Output: John

name = "Alice";
console.log(name); // Output: Alice
```

Rules for Variable Names:

1. Must start with a letter, _, or $.

   - ✅ llet myName = "John";
   - ✅ llet $price = 50;
   - ✅ llet _count = 10;
   - ❌ let 2cool = "Nope"; (Cannot start with a number)

2. Cannot use JavaScript **reserved keywords** like let, class, or function.

Special Notes:

1. **let vs var:**
   - let is block-scoped (works only inside {} where it's defined).
   - var is function-scoped (accessible throughout the function).

   Example:

```
{
  let x = 10;
  console.log(x); // Output: 10
}
// console.log(x); // Error: x is not defined (because of block scope)
```

```
{
  var y = 20;
  console.log(y); // Output: 20
}
console.log(y); // Output: 20 (var is accessible outside the block)
```

2. **const for Constant Data:**

   Use const for values that don’t change, like mathematical constants or configuration settings.

   Example:

   ```
   const companyName = "TechCorp";
   // companyName = "NewName"; // Error: Assignment to constant variable
   ```

Practical Example:

Simple Program Using Variables:

```
let name = "Alice"; // Store a name
let age = 28;       // Store age
const country = "USA"; // Country won't change

console.log(`Hi, my name is ${name}. I am ${age} years old and live in ${country}.`);
// Output: Hi, my name is Alice. I am 28 years old and live in USA.

age = 29; // Update age
console.log(`Next year, I will be ${age}.`);
// Output: Next year, I will be 29.
```

5. OPERATORS IN JAVASCRIPT:

Operators are special symbols or keywords used to perform operations on variables and values. They are essential for manipulating data, making decisions, and building logical expressions. Here's a breakdown of the most important operators in JavaScript, explained simply and with examples.

1. Arithmetic Operators

These operators perform basic mathematical operations:

- **Addition (+):** Adds two numbers.

  Example:

  ```
  let sum = 5 + 3;  // 8
  ```

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

  Example:

  ```
  let difference = 10 - 4;  // 6
  ```

- **Multiplication (*):** Multiplies two numbers.

  Example:

  ```
  let product = 7 * 6;  // 42
  ```

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

  Example:

  ```
  let quotient = 20 / 5;  // 4
  ```

- **Modulus (%):** Returns the remainder of a division.

  Example:

```
let remainder = 15 % 4;  // 3
```

- **Increment (++):** Increases a variable's value by one.
  Example:

  ```
  let x = 5;
  x++;  // x is now 6
  ```

- **Decrement (--):** Decreases a variable's value by one.
  Example:

  ```
  let y = 8;
  y--;  // y is now 7
  ```

2. Assignment Operators

These operators assign values to variables:

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

  ```
  let a = 10;  // a is 10
  ```

- **+= (Addition assignment):** Adds the right operand to the left operand and assigns the result.
  Example:

  ```
  let b = 5;
  b += 3;  // b is now 8
  ```

- **-= (Subtraction assignment):** Subtracts the right operand from the left operand and assigns the result.
  Example:

  ```
  let c = 10;
  c -= 4;  // c is now 6
  ```

- ***= (Multiplication assignment):** Multiplies the variable by the right operand and assigns the result.
  Example:

  ```
  let d = 7;
  d *= 2;  // d is now 14
  ```

- **/= (Division assignment):** Divides the variable by the right operand and assigns the result.
  Example:

  ```
  let e = 20;
  e /= 4;  // e is now 5
  ```

- **%= (Modulus assignment):** Calculates the modulus and assigns the result.
  Example:

  ```
  let f = 15;
  f %= 4;  // f is now 3
  ```

3. Comparison Operators

These operators compare values and return a boolean (true or false):

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

  Example:

  ```
  let isEqual = (5 == '5');  // true
  ```

- **=== (Strict equal to):** Checks if two values are equal and of the same type.

  Example:

  ```
  let isStrictEqual = (5 === '5');  // false
  ```

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

  Example:

  ```
  let isNotEqual = (5 != '5');  // false
  ```

- **!== (Strict not equal to):** Checks if two values are not equal and/or not of the same type.

  Example:

  ```
  let isStrictNotEqual = (5 !== '5');  // true
  ```

- **> (Greater than):** Checks if the value on the left is greater than the value on the right.

  Example:

  ```
  let isGreater = (10 > 7);  // true
  ```

- **< (Less than):** Checks if the value on the left is less than the value on the right.

  Example:

  ```
  let isLess = (3 < 8);  // true
  ```

- **>= (Greater than or equal to):** Checks if the value on the left is greater than or equal to the value on the right.
  Example:

  ```
  let isGreaterOrEqual = (5 >= 5);  // true
  ```

- **<= (Less than or equal to):** Checks if the value on the left is less than or equal to the value on the right.
  Example:

  ```
  let isLessOrEqual = (9 <= 10);  // true
  ```

4. Logical Operators

These operators combine or invert boolean values:

- **&& (Logical AND):** Returns true if both expressions are true.
  Example:

  ```
  let andResult = (5 > 3 && 10 < 15);  // true
  ```

- **|| (Logical OR):** Returns true if at least one of the expressions is true.
  Example:

  ```
  let orResult = (5 < 3 || 10 < 15);  // true
  ```

- **! (Logical NOT):** Inverts the boolean value of the expression.
  Example:

  ```
  let notResult = !(5 > 3);  // false
  ```

5. String Operators

These operators are used to manipulate string values:

- **+ (Concatenation):** Combines two or more strings.
  Example:

  ```
  let greeting = "Hello" + " " + "World!";  // "Hello World!"
  ```

- **+= (Concatenation assignment):** Adds and assigns a string to an existing string variable.
  Example:

  ```
  let message = "Hello";
  message += " World!";  // "Hello World!"
  ```

6. Other Operators

- **Ternary (Conditional) Operator (?:):** A shorthand for if...else.
  Example:

  ```
  let age = 18;
  let canVote = (age >= 18) ? "Yes" : "No";  // "Yes"
  ```

- **typeof Operator:** Returns the type of a variable.
  Example:

  ```
  let type = typeof "Hello";  // "string"
  ```

- **instanceof Operator:** Checks if an object is an instance of a specific class or constructor function.
  Example:

```
let date = new Date();
let isDate = date instanceof Date;  // true
```

- **Comma Operator (,)**: Evaluates multiple expressions and returns the result of the last expression.
  Example:

```
let result = (5, 10, 15);  // 15
```

6. DATA TYPES IN JAVASCRIPT:

In JavaScript, **data types** define what kind of value a variable can hold. Understanding the different data types is important for working with variables, performing operations, and handling values in your code. JavaScript has two types of data types: **Primitive** and **Non-Primitive** (also called Reference types). Here's an easy way to understand them.

1. Primitive Data Types

Primitive data types are the most basic types of data. They are immutable, meaning their values cannot be changed once assigned.

1.1. Number

Numbers represent both integer and floating-point numbers.

- **Example:**

```
let age = 25;     // Integer
let price = 19.99; // Floating point number
```

1.2. String

Strings are used to store text or a sequence of characters.

- **Example:**

```
let name = "John";  // String
let greeting = 'Hello, World!'; // String
```

1.3. Boolean

Booleans are used to represent two values: true or false.

- **Example:**

```
let isJavaScriptFun = true;  // Boolean (true)
let isOlderThan18 = false;   // Boolean (false)
```

1.4. Undefined

A variable that is declared but not assigned a value is automatically assigned the value undefined.

- **Example:**

```
let car;
console.log(car); // undefined
```

1.5. Null

Null represents the intentional absence of any value. It is used to indicate that a variable is empty.

- **Example:**

```
let result = null;  // null represents no value
```

1.6. Symbol **(ES6)**

Symbols are unique, immutable values, often used for object properties that are guaranteed to be unique.

- **Example:**

```
let sym = Symbol('description');  // Creating a unique symbol
```

1.7. BigInt **(ES11)**

BigInt is used to represent large integers that cannot be handled by the Number data type.

- **Example:**

```
let bigNumber = 1234567890123456789012345678901234567890n;  // BigInt
```

2. Non-Primitive (Reference) Data Types

Non-primitive data types are more complex types. These types can store collections of data and are mutable (their content can be changed).

2.1. Object

An object is a collection of key-value pairs, where keys are strings (or Symbols), and values can be any data type (primitive or non-primitive).

- **Example:**

```
let person = {
  name: "Alice",
  age: 30,
  isStudent: false
};
```

2.2. Array

Arrays are special types of objects used to store ordered collections of data. Each element in an array has an index starting from 0.

- **Example:**

```
let fruits = ["Apple", "Banana", "Cherry"];  // Array of strings
let numbers = [1, 2, 3, 4, 5];  // Array of numbers
```

2.3. Function

Functions are also objects in JavaScript. A function is a block of code that can be executed and can return a value.

- **Example:**

```
function greet(name) {
  return "Hello, " + name;
}
console.log(greet("John"));  // Output: "Hello, John"
```

3. Type Conversion

Sometimes, JavaScript automatically converts between different data types, which is called **type coercion**. You can also convert types manually using functions like String(), Number(), or Boolean().

- **Example of type coercion:**

```
let num = 10;
let str = "5";
console.log(num + str); // Output: "105" (number is coerced to string)
```

- **Example of explicit type conversion:**

```
let numString = "123";
let num = Number(numString);  // Convert string to number
console.log(num);  // Output: 123 (as a number)
```

7. CONDITIONAL STATEMENT IN JAVASCRIPT:

Conditional statements are used in JavaScript to perform different actions based on different conditions. These statements allow you to make decisions in your code, helping your program to behave differently under different situations. The most common conditional statements in JavaScript are **if**, **else**, **else if**, and **switch**.

1. The **if** Statement

The if statement is used to execute a block of code only if a specified condition is true. If the condition evaluates to true, the code inside the block will run.

Syntax:

```
if (condition) {
   // Code to be executed if the condition is true
}
```

Example:

```
let age = 18;
```

```
if (age >= 18) {
    console.log("You are an adult.");
}
```

Output:

```
You are an adult.
```

In this example, the code inside the if block runs only because the condition age >= 18 evaluates to true.

2. The **else** Statement

The else statement is used alongside the if statement to specify a block of code that will be executed if the condition is false.

Syntax:

```
if (condition) {
    // Code to be executed if the condition is true
} else {
    // Code to be executed if the condition is false
}
```

Example:

```
let age = 16;
if (age >= 18) {
    console.log("You are an adult.");
} else {
    console.log("You are a minor.");
}
```

Output:

```
You are a minor.
```

Here, since age is less than 18, the code inside the else block is executed.

3. The **else if** Statement

The else if statement allows you to check multiple conditions. It comes after the if statement and is followed by one or more conditions.

Syntax:

```
if (condition1) {
   // Code to be executed if condition1 is true
} else if (condition2) {
   // Code to be executed if condition2 is true
} else {
   // Code to be executed if both conditions are false
}
```

Example:

```
let age = 20;
if (age < 18) {
   console.log("You are a minor.");
} else if (age >= 18 && age <= 65) {
   console.log("You are an adult.");
} else {
    console.log("You are a senior.");
}
```

Output:

```
You are an adult.
```

In this case, since age is between 18 and 65, the else if block is executed.

4. The **switch** Statement

The switch statement is used to check a variable or expression against a list of possible values. It is an alternative to multiple else if statements and can be more readable when dealing with many conditions.

Syntax:

```
switch (expression) {
    case value1:
        // Code to be executed if expression equals value1
        break;
    case value2:
        // Code to be executed if expression equals value2
        break;
    default:
        // Code to be executed if no case matches
}
```

Example:

```
let day = 3;
```

```
switch (day) {
    case 1:
        console.log("Monday");
        break;
    case 2:
        console.log("Tuesday");
        break;
    case 3:
        console.log("Wednesday");
        break;
    case 4:
        console.log("Thursday");
        break;
    case 5:
        console.log("Friday");
        break;
    case 6:
        console.log("Saturday");
        break;
    case 7:
        console.log("Sunday");
        break;
    default:
        console.log("Invalid day");
}
```

Output:

```
Wednesday
```

In this example, the value of day is 3, so the case 3 block is executed, and "Wednesday" is printed. The break statement is used to stop further checking once a match is found.

5. Ternary Operator (Conditional Operator)

The ternary operator is a shorthand for an if-else statement. It evaluates a condition and returns one of two values depending on whether the condition is true or false.

Syntax:

```
condition ? value_if_true : value_if_false;
```

Example:

```
let age = 18;
let message = (age >= 18) ? "You are an adult." : "You are a minor.";
console.log(message);
```

Output:

```
You are an adult.
```

This is a compact version of the if-else statement. If age is 18 or more, the first message is printed; otherwise, the second message is printed.

7. LOOPS IN JAVASCRIPT:

Loops are used to repeat a block of code multiple times in JavaScript. They are essential for automating repetitive tasks and making your code more efficient. JavaScript provides several types of loops, each suitable for different situations. The most common loops are **for**, **while**, **do...while**, and **for...in**/ **for...of** loops.

1. The **for** Loop

The for loop is the most commonly used loop in JavaScript. It repeats a block of code a specified number of times. It is especially useful when you know how many times you want the loop to run.

Syntax:

```
for (initialization; condition; increment/decrement) {
    // Code to be executed
}
```

- **Initialization**: Typically used to set a starting value for the counter.
- **Condition**: The loop runs as long as this condition is true.
- **Increment/Decrement**: Updates the counter each time the loop runs.

Example:

```
for (let i = 0; i < 5; i++) {
    console.log(i); // Prints numbers from 0 to 4
}
```

Output:

```
0
1
2
3
4
```

In this example, the for loop runs 5 times, starting from i = 0 and running as long as i < 5. After each iteration, i is incremented by 1.

2. The **while** Loop

The while loop is used when you want to repeat a block of code while a specified condition is true. The loop will continue as long as the condition evaluates to true.

Syntax:

```
while (condition) {
   // Code to be executed
}
```

Example:

```
let i = 0;
while (i < 5) {
   console.log(i); // Prints numbers from 0 to 4
   i++;
}
```

Output:

```
0
1
2
3
4
```

In this example, the while loop continues to execute as long as i is less than 5. The counter i is manually incremented inside the loop.

3. The **do...while** Loop

The do...while loop is similar to the while loop, but it guarantees that the code inside the loop will be executed at least once, even if the condition is false initially.

Syntax:

```
do {
   // Code to be executed
} while (condition);
```

Example:

```
let i = 0;
do {
   console.log(i); // Prints numbers from 0 to 4
   i++;
} while (i < 5);
```

Output:

```
0
1
2
3
4
```

Here, the code inside the do block is executed first, and then the condition i < 5 is checked. If true, it continues; otherwise, it stops.

4. The **for...in** Loop

The for...in loop is used to iterate over the properties of an object. It loops through the keys (or property names) of an object, one by one.

Syntax:

```
for (let key in object) {
  // Code to be executed
}
```

Example:

```
let person = {
  name: "John",
  age: 30,
  city: "New York"
};

for (let key in person) {
  console.log(key + ": " + person[key]); // Prints each property name and value
}
```

Output:

```
name: John
age: 30
city: New York
```

In this example, the for...in loop iterates over each property of the person object and logs both the property name (key) and its corresponding value (person[key]).

5. The **for...of** Loop

The for...of loop is used to iterate over iterable objects like arrays, strings, and other collections. It is a simpler way to iterate through values in an array or any iterable object.

Syntax:

```
for (let item of iterable) {
    // Code to be executed
}
```

Example:

```
let numbers = [10, 20, 30, 40, 50];
for (let num of numbers) {
    console.log(num); // Prints each number in the array
}
```

Output:

```
10
20
30
40
50
```

In this example, the for...of loop iterates over each element in the numbers array and prints each number one by one.

6. Breaking Out of Loops

Sometimes, you may want to stop the loop before it completes all iterations. This can be done using the break statement, which immediately exits the loop.

Example:

```
for (let i = 0; i < 5; i++) {
    if (i === 3) {
        break; // Exits the loop when i equals 3
    }
    console.log(i); // Prints 0, 1, 2
}
```

Output:

```
0
1
2
```

Here, the loop stops when i reaches 3 because of the break statement.

7. Skipping an Iteration

If you want to skip the current iteration of the loop and move to the next one, you can use the continue statement.

Example:

```
for (let i = 0; i < 5; i++) {
    if (i === 2) {
        continue; // Skips the iteration when i equals 2
    }
    console.log(i); // Prints 0, 1, 3, 4
}
```

Output:

```
0
1
3
4
```

In this case, when i equals 2, the continue statement skips that iteration and proceeds to the next.

9. TEMPLATE LITERALS IN JAVASCRIPT:

Template literals (also known as template strings) are a feature in JavaScript introduced in ES6 that allows you to work with strings more efficiently. Unlike regular strings, template literals provide more powerful features, such as string interpolation, multi-line strings, and embedded expressions, making your code cleaner and easier to understand.

1. What Are Template Literals?

Template literals are strings that are enclosed in backticks (`) instead of single or double quotes. This allows you to embed expressions inside the string and also write multi-line strings without needing special syntax.

Syntax:

```
let message = `Your string here`;
```

2. String Interpolation (Embedding Expressions)

One of the most useful features of template literals is **string interpolation**, which allows you to embed variables or expressions directly within the string. This makes it easy to combine strings and variables without having to use concatenation.

You can embed expressions inside ${}.

Example:

```
let name = "John";
let age = 30;
let greeting = `Hello, my name is ${name} and I am ${age} years old.`;
console.log(greeting);
```

Output:

```
Hello, my name is John and I am 30 years old.
```

In this example, the variables name and age are inserted directly into the string using ${}. This is much cleaner and more readable than traditional string concatenation.

3. Multi-line Strings

Template literals also support **multi-line strings**, which is especially useful when you want to include line breaks in your strings. With regular strings (using single or double quotes), you need to use escape characters (\n) to insert new lines. With template literals, this is not necessary.

Example:

```
let message = `This is a string
that spans multiple lines
without using escape characters.`;
```

```
console.log(message);
```

Output:

```
This is a string
that spans multiple lines
without using escape characters.
```

The string is displayed exactly as written between the backticks, preserving the line breaks.

4. Expression Embedding

Template literals allow you to embed any valid JavaScript expression inside ${}. This means you can perform calculations or call functions directly within the string.

Example:

```
let num1 = 5;
let num2 = 10;
let sum = `The sum of ${num1} and ${num2} is ${num1 + num2}.`;
console.log(sum);
```

Output:

```
The sum of 5 and 10 is 15.
```

Here, the expression ${num1 + num2} is evaluated and the result (15) is inserted into the string.

5. Tagged Template Literals

Tagged template literals are an advanced feature in JavaScript where you can create custom functions (called "tags") that process the template literal's content. A tag function can manipulate the template literal’s content before it is returned.

Syntax:

```
tagName`template literal`;
```

Example:

```
function greet(strings, name, age) {
   return `${strings[0]}${name}${strings[1]}${age}${strings[2]}`;
}


let name = "Alice";
let age = 25;
let message = greet`Hello, my name is ${name} and I am ${age} years old.`;
console.log(message);
```

Output:

```
Hello, my name is Alice and I am 25 years old.
```

In this example, the greet function is a tag that takes the string parts and expressions of the template literal and returns the processed result.

6. Nested Template Literals

You can use template literals inside other template literals, allowing for more complex string formatting.

Example:

```
let firstName = "John";
let lastName = "Doe";
let fullName = `Full Name: ${firstName} ${lastName}`;
let message = `Hello, ${fullName}`;
console.log(message);
```

Output:

```
Hello, Full Name: John Doe
```

Here, the fullName is embedded inside another template literal.

7. Practical Use Cases of Template Literals

- **Dynamic HTML generation**: When creating HTML elements dynamically in JavaScript, template literals are extremely useful.

  Example:

  ```
  let title = "Welcome!";
  let content = "This is a dynamic content.";
  let html = `
    <div>
      <h1>${title}</h1>
      <p>${content}</p>
    </div>
  `;
  document.body.innerHTML = html;
  ```

- **Complex strings**: You can easily create complex strings that include calculations, dynamic content, and expressions.

10. OBJECTS AND ARRAYS IN JAVASCRIPT:

In JavaScript, **objects** and **arrays** are two of the most commonly used data structures. They allow you to store and manage multiple values and organize your data effectively. Let's explore each of them in simple terms with examples.

1. Objects in JavaScript

What is an Object?

An **object** in JavaScript is a collection of key-value pairs, where each key (or property) is a string, and the value can be any type of data (such as a number, string, array, or even another object). Objects are used to store related data and are an essential part of JavaScript.

Creating an Object

To create an object, we use curly braces {} and define properties in the format key: value.

Example:

```
let person = {
   name: "John",
   age: 30,
   isStudent: false
};
console.log(person);
```

Output:

```
{ name: "John", age: 30, isStudent: false }
```

In this example, name, age, and isStudent are the keys (or properties), and "John", 30, and false are the corresponding values.

Accessing Object Properties

You can access an object's properties using either dot notation or bracket notation.

Dot Notation:

```
console.log(person.name); // "John"
```

Bracket Notation:

```
console.log(person['age']); // 30
```

Note: Bracket notation is useful when the property name is dynamic or contains spaces or special characters.

Modifying Object Properties

You can modify an object’s properties by assigning new values.

Example:

```
person.age = 35; // Modify age
console.log(person.age); // 35
```

You can also add new properties to an object.

Example:

```
person.country = "USA"; // Add new property
console.log(person.country); // "USA"
```

Methods in Objects

Objects can also contain **methods**, which are functions associated with an object.

Example:

```
let car = {
   brand: "Toyota",
   model: "Corolla",
   drive: function() {
      console.log("The car is driving!");
   }
};

car.drive(); // "The car is driving!"
```

2. Arrays in JavaScript

What is an Array?

An **array** in JavaScript is a list-like object that can store multiple values in a single variable. Arrays can hold elements of any type, including numbers, strings, and even other arrays or objects.

Creating an Array

Arrays are created using square brackets [] with elements separated by commas.

Example:

```
let fruits = ["apple", "banana", "cherry"];
console.log(fruits);
```

Output:

```
 ["apple", "banana", "cherry"]
```

Accessing Array Elements

You can access elements in an array using their index. Array indexes start at 0.

Example:

```
console.log(fruits[0]); // "apple"
console.log(fruits[1]); // "banana"
```

Modifying Array Elements

You can change an element in an array by using its index.

Example:

```
fruits[1] = "blueberry"; // Modify the second element
console.log(fruits); // ["apple", "blueberry", "cherry"]
```

Array Methods

Arrays have built-in methods for manipulating the elements. Here are some common ones:

Push and Pop:

- **push()**: Adds an element to the end of the array.
- **pop()**: Removes the last element from the array.

```
fruits.push("orange"); // Add "orange" to the end
console.log(fruits); // ["apple", "blueberry", "cherry", "orange"]

fruits.pop(); // Remove the last element ("orange")
console.log(fruits); // ["apple", "blueberry", "cherry"]
```

Shift and Unshift:

- **shift()**: Removes the first element from the array.

- **unshift()**: Adds an element to the beginning of the array.

```
fruits.shift(); // Remove the first element ("apple")
console.log(fruits); // ["blueberry", "cherry"]

fruits.unshift("kiwi"); // Add "kiwi" to the beginning
console.log(fruits); // ["kiwi", "blueberry", "cherry"]
```

ForEach:

The **forEach()** method allows you to execute a function for each element in the array.

```
fruits.forEach(function(fruit) {
   console.log(fruit);
});
```

Output:

```
kiwi
blueberry
cherry
```

Map:

The **map()** method creates a new array by applying a function to each element in the original array.

```
let lengths = fruits.map(function(fruit) {
   return fruit.length;
});
console.log(lengths); // [4, 9, 6]
```

11. FUNCTIONS IN JAVASCRIPT:

A **function** in JavaScript is a block of reusable code designed to perform a specific task. Functions allow you to break down complex problems into smaller, manageable parts, making your code more modular, readable, and maintainable.

1. What is a Function?

A **function** is a way to define a set of instructions that can be executed whenever you call the function. It helps to avoid repetition, making code more efficient and organized.

Function Syntax:

```
function functionName(parameters) {
   // Code to execute
   return result;  // Optional return statement
}
```

- function is the keyword to define a function.
- functionName is the name of the function.
- parameters are the inputs that the function uses (optional).
- The function’s **body** contains the instructions that run when the function is called.
- return is used to send back a value from the function (optional).

2. Example of a Simple Function

Here's an example of a simple function that adds two numbers:

Example:

```
function addNumbers(a, b) {
   return a + b;  // Return the sum of a and b
}
```

```
let result = addNumbers(3, 4);  // Call the function with arguments 3 and 4
console.log(result);  // Output: 7
```

In this example:

- The function addNumbers takes two parameters (a and b).
- It returns the sum of these two numbers.
- The result is stored in the variable result, which is then printed to the console.

3. Function Parameters and Arguments

- **Parameters** are the names used when defining a function.
- **Arguments** are the actual values passed to the function when it is called.

In the example above, a and b are **parameters**. When we call addNumbers(3, 4), the values 3 and 4 are the **arguments** passed to the function.

Default Parameters:

JavaScript allows you to set **default values** for parameters if no argument is provided.

Example:

```
function greet(name = "Guest") {
   console.log("Hello, " + name + "!");
}

greet();          // Output: Hello, Guest!
greet("John");    // Output: Hello, John!
```

In this case, the name parameter has a default value of "Guest", so if no argument is passed, it uses this default value.

4. Returning Values from Functions

A function can return a value to the code that called it. The return statement specifies the value that should be sent back.

Example:

```
function multiply(x, y) {
   return x * y;  // Multiply and return the result
}

let result = multiply(5, 6);
console.log(result);  // Output: 30
```

In this example, the function returns the product of x and y, and the result is stored in the variable result.

5. Function Expressions

In addition to defining functions using the function keyword, you can also create functions using **function expressions**. This means that the function is assigned to a variable.

Example:

```
let square = function(x) {
   return x * x;
};

console.log(square(4));  // Output: 16
```

In this example, the function is defined and assigned to the square variable. You can then call it like a regular function.

6. Arrow Functions (ES6)

Arrow functions are a shorter way to write functions in JavaScript. They were introduced in ECMAScript 6 (ES6) and are commonly used in modern JavaScript development.

Syntax:

```
const functionName = (parameters) => {
    // Code to execute
    return result;
};
```

Example:

```
const add = (a, b) => a + b;  // Shorter syntax for the function
console.log(add(3, 5));  // Output: 8
```

In this example, add is an arrow function that adds two numbers. Arrow functions are often preferred for their simplicity, especially when the function body is simple.

7. Function Scope

In JavaScript, **scope** refers to the context in which a variable or function is accessible. Variables declared inside a function are **local** to that function and cannot be accessed outside it.

Example:

```
function greet() {
    let greeting = "Hello, world!";  // Local variable
    console.log(greeting);  // This will work
}

greet();
```

console.log(greeting); // This will result in an error, because greeting is not accessible outside the function.

In this example, the variable greeting is defined inside the function greet, so it is not accessible outside the function.

8. Function Hoisting

In JavaScript, function declarations are **hoisted**. This means they are moved to the top of their scope during the compilation phase, allowing you to call a function before it is defined in the code.

Example:

```
greet();  // This will work, even before the function is defined.

function greet() {
   console.log("Hello!");
}
```

However, **function expressions** are not hoisted. So, trying to call a function defined with a function expression before it's declared will result in an error.

Example of function expression hoisting error:

```
sayHello();  // This will cause an error

let sayHello = function() {
   console.log("Hello!");
};
```

9. Callback Functions

A **callback function** is a function that is passed as an argument to another function and is executed after the completion of that function.

Example:

```
function greet(name, callback) {
    console.log("Hello, " + name + "!");
    callback();  // Call the callback function after greeting
}

function farewell() {
    console.log("Goodbye!");
}

greet("John", farewell);  // Output: "Hello, John!" followed by "Goodbye!"
```

In this example, the greet function accepts a callback (farewell), and it is called after the greeting is printed.

10. Recursion

Recursion is a programming technique where a function calls itself in order to solve a problem.

Example:

```
function factorial(n) {
    if (n === 0) {
        return 1;  // Base case
    }
    return n * factorial(n - 1);  // Recursive call
}
```

```
console.log(factorial(5));  // Output: 120
```

In this example, the factorial function calculates the factorial of a number by calling itself until it reaches the base case (n === 0).

12. EVENTS IN JAVASCRIPT:

1. What is an Event?

An **event** in JavaScript is an action or occurrence that the browser recognizes, such as a mouse click, keyboard press, page load, or form submission. Events are used to interact with the user and trigger functions when certain conditions are met.

For example, when a user clicks a button on a webpage, a "click" event occurs, and you can set up JavaScript to respond to that event.

2. Common Types of Events

There are many different types of events in JavaScript. Some of the most common events include:

User Interaction Events:

- **click**: Occurs when an element is clicked.
- **dblclick**: Occurs when an element is double-clicked.
- **mouseover**: Occurs when the mouse pointer is moved over an element.
- **mouseout**: Occurs when the mouse pointer leaves an element.
- **keydown**: Occurs when a key is pressed down.
- **keyup**: Occurs when a key is released.
- **input**: Occurs when a user types in an input field.
- **focus**: Occurs when an element gains focus (e.g., when a user clicks on a text field).

- **blur**: Occurs when an element loses focus.

Page Events:

- **load**: Occurs when the page has finished loading.
- **unload**: Occurs when the page is about to be unloaded.
- **resize**: Occurs when the window is resized.
- **scroll**: Occurs when the page is scrolled.

Form Events:

- **submit**: Occurs when a form is submitted.
- **change**: Occurs when the value of an input element is changed.

3. How to Handle Events

There are three main ways to handle events in JavaScript:

1. **Inline Event Handlers**
2. **Event Listener**
3. **Event Handler Property**

3.1 Inline Event Handlers

You can define events directly within HTML elements using attributes. This is the simplest method but is generally not recommended for large projects due to separation of concerns.

Example:

```
<button onclick="alert('Button clicked!')">Click Me</button>
```

In this example, when the user clicks the button, the event handler onclick triggers a JavaScript alert().

3.2 Event Listener

The most modern and preferred way of handling events is by using addEventListener(). This method allows you to attach an event handler to an element, enabling multiple event listeners to be added to the same element.

Syntax:

```
element.addEventListener(event, function, useCapture);
```

- event: The type of event (e.g., click, mouseover).
- function: The function to run when the event is triggered.
- useCapture: Optional, specifies whether the event should be captured during the capture phase.

Example:

```
<button id="myButton">Click Me</button>


<script>
   let button = document.getElementById("myButton");


   button.addEventListener("click", function() {
      alert("Button clicked!");
   });
</script>
```

In this example, when the user clicks the button, an alert box will appear, showing the message "Button clicked!".

3.3 Event Handler Property

You can also assign an event handler directly to an element using its event handler properties. This method is less flexible than using addEventListener() because you can only have one event listener for each event type.

Example:

```
<button id="myButton">Click Me</button>

<script>
   let button = document.getElementById("myButton");

   button.onclick = function() {
      alert("Button clicked!");
   };
</script>
```

In this example, when the button is clicked, the assigned function will run and show an alert.

4. Event Object

When an event occurs, the browser automatically passes an event object to the event handler. This object contains information about the event, such as which element triggered the event and any additional data (e.g., mouse coordinates or key code).

Example:

```
<button id="myButton">Click Me</button>

<script>
   let button = document.getElementById("myButton");

   button.addEventListener("click", function(event) {
      console.log(event);  // Logs the event object
```

```
        console.log("Button clicked!");
    });
</script>
```

In this example, the event object is logged when the button is clicked, providing details such as the mouse position, the type of event, and the target element.

5. Event Propagation

In JavaScript, events propagate through the DOM in two phases:

1. **Capturing Phase** (Event travels down from the root to the target element).
2. **Bubbling Phase** (Event travels up from the target element to the root).

You can control how an event propagates by using stopPropagation() and preventDefault().

- **stopPropagation()**: Stops the event from propagating (bubbling or capturing).
- **preventDefault()**: Prevents the default action associated with the event.

Example:

```
<div id="parentDiv">
    <button id="childButton">Click Me</button>
</div>


<script>
    document.getElementById("parentDiv").addEventListener("click", function(event) {
        alert("Parent Div clicked!");
        event.stopPropagation();  // Stop the event from bubbling
    });

    document.getElementById("childButton").addEventListener("click", function() {
        alert("Button clicked!");
    });
</script>
```

In this example:

- When you click the button, the click event triggers on the button first.
- Then, the event would bubble to the parent div, but stopPropagation() prevents it, so only the button’s alert will show.

6. Removing Event Listeners

You can remove an event listener from an element using the removeEventListener() method. This is useful if you no longer want the event to trigger after a certain condition is met.

Example:

```
<button id="myButton">Click Me</button>


<script>
    let button = document.getElementById("myButton");


    function handleClick() {
        alert("Button clicked!");
    }


    button.addEventListener("click", handleClick);


    // Remove the event listener after 3 seconds
    setTimeout(function() {
        button.removeEventListener("click", handleClick);
    }, 3000);
```

```
</script>
```

In this example, the click event listener is removed after 3 seconds, so after that, clicking the button will not trigger the alert.

7. Event Delegation

Event delegation is a technique where you attach a single event listener to a parent element, and the event listener will handle events for child elements as well. This is useful when you have multiple child elements, and you don't want to attach an individual event listener to each one.

Example:

```
<ul id="myList">
   <li>Item 1</li>
   <li>Item 2</li>
   <li>Item 3</li>
</ul>


<script>
   let list = document.getElementById("myList");


   list.addEventListener("click", function(event) {
      if (event.target.tagName === "LI") {
         alert("List item clicked: " + event.target.textContent);
      }
   });
</script>
```

In this example, a single click event listener is attached to the ul element, but it listens for clicks on any of the li elements. This is an example of **event delegation**.

13. DOM MANIPULATION IN JAVASCRIPT:

1. What is DOM Manipulation?

DOM manipulation refers to the process of accessing and modifying the elements of a web page using JavaScript. Through DOM manipulation, you can:

- Change text content of elements
- Add or remove HTML elements
- Modify styles of elements
- Respond to user interactions like clicks or key presses

In simple terms, DOM manipulation allows you to update the structure and appearance of a webpage after it has loaded.

2. Common Methods for DOM Manipulation

2.1 Selecting Elements

To manipulate elements in the DOM, you first need to select them. JavaScript provides several methods for selecting elements:

getElementById()

Selects an element by its unique id attribute.

```
let element = document.getElementById("myElement");
```

getElementsByClassName()

Selects elements by their class name. It returns a live HTMLCollection (array-like object).

```
let elements = document.getElementsByClassName("myClass");
```

getElementsByTagName()

Selects elements by their tag name (e.g., div, p, button).

```
let elements = document.getElementsByTagName("div");
```

querySelector()

Selects the first matching element using a CSS selector.

```
let element = document.querySelector(".myClass");  // Selects the first element with class "myClass"
```

querySelectorAll()

Selects all matching elements using a CSS selector, returning a static NodeList.

```
let elements = document.querySelectorAll(".myClass");  // Selects all elements with class "myClass"
```

2.2 Modifying Content

After selecting an element, you can manipulate its content:

Changing Text Content

You can change the text inside an element using the textContent property:

```
let element = document.getElementById("myElement");
element.textContent = "New Text Content";
```

Changing HTML Content

To modify the HTML inside an element, use the innerHTML property:

```
let element = document.getElementById("myElement");
element.innerHTML = "<strong>New HTML Content</strong>";
```

Changing Attribute Values

You can modify an element's attributes (like src, href, etc.) using setAttribute():

```
let image = document.getElementById("myImage");
image.setAttribute("src", "newImage.jpg");
```

To get an attribute's value, use getAttribute():

```
let srcValue = image.getAttribute("src");
console.log(srcValue);  // Prints the current src value
```

2.3 Modifying Styles

You can change the styles of elements dynamically by modifying the style property:

Changing Inline Styles

You can change an element's inline styles directly using JavaScript:

```
let element = document.getElementById("myElement");
element.style.color = "blue";  // Changes the text color to blue
element.style.backgroundColor = "yellow";  // Changes the background color to yellow
```

2.4 Adding/Removing Elements

You can add, remove, or modify the elements in the DOM.

Adding New Elements

To add new elements, use createElement() to create the new element and appendChild() or insertBefore() to insert it into the DOM.

```
let newElement = document.createElement("p");
newElement.textContent = "This is a new paragraph!";
let parentElement = document.getElementById("parentDiv");
parentElement.appendChild(newElement);
```

Removing Elements

To remove an element from the DOM, use the removeChild() method:

```
let element = document.getElementById("myElement");
element.parentNode.removeChild(element);
```

Alternatively, you can use the remove() method on the element itself:

```
let element = document.getElementById("myElement");
element.remove();  // Removes the element from the DOM
```

2.5 Event Handling

JavaScript allows you to attach event listeners to elements, enabling interaction with the page (e.g., when a user clicks a button or submits a form).

Adding Event Listeners

Use the addEventListener() method to listen for events on elements:

```
let button = document.getElementById("myButton");
button.addEventListener("click", function() {
   alert("Button clicked!");
});
```

Removing Event Listeners

You can remove an event listener using the removeEventListener() method:

```
button.removeEventListener("click", function() {
    alert("Button clicked!");
});
```

3. Practical Example of DOM Manipulation

Let’s look at a simple example where we manipulate the DOM by adding a new element, changing the content, and responding to user interaction.

HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM Manipulation Example</title>
</head>
<body>
    <div id="parentDiv">
        <button id="addButton">Add New Item</button>
    </div>

    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js):

```
// Get the button and the parent div
let addButton = document.getElementById("addButton");
let parentDiv = document.getElementById("parentDiv");
```

```
// Add an event listener to the button
addButton.addEventListener("click", function() {
    // Create a new list item
    let newItem = document.createElement("p");
    newItem.textContent = "This is a new item!";

    // Add a style to the new item
    newItem.style.color = "green";

    // Append the new item to the parent div
    parentDiv.appendChild(newItem);
});
```

In this example:

- When the "Add New Item" button is clicked, a new <p> element is created.
- The new paragraph has the text "This is a new item!" and is styled with the color green.
- The new item is then appended to the parentDiv on the webpage.

14. PROMISES AND ASYNC/AWAIT IN JAVASCRIPT:

1. Promises

A **Promise** is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

1.1 States of a Promise

A Promise can exist in one of the following states:

1. **Pending**: The initial state, neither fulfilled nor rejected.

2. **Fulfilled**: The operation was successful, and a result is available.
3. **Rejected**: The operation failed, and an error reason is available.

1.2 Creating a Promise

You can create a Promise using the Promise constructor, which accepts a function with two parameters: resolve (for success) and reject (for failure).

Example: Basic Promise

```
let myPromise = new Promise((resolve, reject) => {
   let success = true; // Simulate a condition
   if (success) {
      resolve("Promise fulfilled!");
   } else {
      reject("Promise rejected!");
   }
});

// Using the Promise
myPromise
   .then(result => console.log(result)) // Logs "Promise fulfilled!" if resolved
   .catch(error => console.log(error)); // Logs "Promise rejected!" if rejected
```

1.3 Promise Chaining

Promises can be chained together for sequential asynchronous operations.

Example: Promise Chaining

```
let fetchData = new Promise((resolve, reject) => {
   setTimeout(() => resolve("Data fetched"), 2000);
});
```

```
fetchData
  .then(result => {
    console.log(result); // Logs "Data fetched"
    return "Processing data...";
  })
  .then(result => console.log(result)) // Logs "Processing data..."
  .catch(error => console.log(error));
```

2. Async/Await

The **Async/Await** syntax is a more concise and readable way to handle Promises. It allows you to write asynchronous code that looks and behaves like synchronous code.

2.1 The **async** Keyword

A function prefixed with async always returns a Promise. If the function explicitly returns a value, it is wrapped in a resolved Promise.

Example: Async Function

```
async function greet() {
  return "Hello!";
}

greet().then(result => console.log(result)); // Logs "Hello!"
```

2.2 The **await** Keyword

The await keyword pauses the execution of an async function until the Promise is resolved or rejected. It can only be used inside async functions.

Example: Using Async/Await

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve("Data fetched"), 2000);
    });
}

async function processData() {
    console.log("Fetching data...");
    let data = await fetchData(); // Waits until the Promise resolves
    console.log(data); // Logs "Data fetched"
}

processData();
```

3. Error Handling with Async/Await

Error handling in async/await is done using try...catch blocks. This makes error handling straightforward and similar to synchronous code.

Example: Error Handling

```
function fetchData(success) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (success) resolve("Data fetched successfully!");
            else reject("Failed to fetch data.");
        }, 2000);
    });
}

async function processData() {
    try {
        console.log("Fetching data...");
        let data = await fetchData(true); // Change to false to simulate an error
        console.log(data);
    } catch (error) {
        console.error(error); // Handles rejection
    }
}

processData();
```

4. Practical Example: Fetching Data from an API

Using Promises

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error("Error:", error));
```

Using Async/Await

```
async function fetchPost() {
    try {
        let response = await fetch("https://jsonplaceholder.typicode.com/posts/1");
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.error("Error:", error);
    }
}
fetchPost();
```

TYPESCRIPT

- TypeScript is a statically-typed, compiled superset of JavaScript.
- It enhances JavaScript by adding static types, which can catch errors at compile time rather than during execution.
- This helps developers catch potential bugs early, leading to faster and more reliable development.

KEY FEATURES OF TYPESCRIPT:

- **Static typing:**

  You can define types for variables, parameters, and return values.
- **Compile-time checks:**

  TypeScript ensures that types are used correctly before running the code.
- **Works everywhere JavaScript runs:**

TypeScript can be compiled to standard JavaScript, which runs in all browsers, environments, and platforms.

❖ **Support for modern JavaScript features:**

TypeScript supports many ES6 and ES2015 features such as classes, modules, and async/await.

DATATYPES:

1. String

❖ Strings represent textual data. In TypeScript, a string is defined using either single quotes (') or double quotes (").
❖ You can store any text in a string variable.

Example:

```
const name: string = 'Alice';

console.log(name); // Prints: Alice
```

2. Number

❖ In TypeScript, all numbers (integers, floating-point values) are represented by the number type.
❖ It also supports decimal, hexadecimal, binary, and octal literals.

Example:

```
const age: number = 30;   // A regular number
```

```
console.log(age); // Prints: 30
```

3. Boolean

A boolean type represents logical values true and false.

Example:

```
const isActive: boolean = true;

console.log(isActive); // Prints: true
```

4. Array

- ❖ Arrays hold multiple values of the same type.
- ❖ TypeScript allows you to specify the type of elements in an array.
- ❖ You can define arrays in two ways:
    - ➢ Using [] (array syntax).
    - ➢ Using Array<type>.

Example 1 (Array using []):

```
const fruits: string[] = ['Apple', 'Banana', 'Cherry'];

console.log(fruits); // Prints: ['Apple', 'Banana', 'Cherry']
```

Example 2 (Array using Array<type>):

```
const numbers: Array<number> = [10, 20, 30];

console.log(numbers); // Prints: [10, 20, 30]
```

5. Tuple

- ❖ Tuples are arrays where each element can have a different type, and their number and order are fixed.
- ❖ For example, a tuple can store a string followed by a number.

Example:

```
const person: [string, number] = ['Alice', 30];

console.log(person); // Prints: ['Alice', 30]
```

6. Any

- ❖ The any type is used when you don’t want to specify a type, or when the type of a value is unknown.
- ❖ It allows variables to hold values of any type, effectively opting out of TypeScript's type-checking.

Example:

```
let data: any = 'Hello';

console.log(data); // Prints: Hello



data = 100;

console.log(data); // Prints: 100



data = { name: 'Alice', age: 30 };

console.log(data); // Prints: { name: 'Alice', age: 30 }
```

7. Enum

- An enum is a way to define a set of named constants.
- Each member of the enum is automatically assigned a numeric value, starting from 0 by default.
- You can customize the starting value or assign values manually.

Example:

```
enum Color {

Red = 1,   // 1

Green,     // 2

Blue,      // 3

}
```

```
const myColor: Color = Color.Green;

console.log(myColor); // Prints: 2
```

8. Void

- The void type is used for functions that don’t return a value.
- It's similar to null or undefined, but it's used specifically for functions.

Example:

```
function logMessage(message: string): void {

console.log(message);

}

logMessage('Hello, TypeScript!'); // Prints: Hello, TypeScript!
```

<u>FUNCTIONS:</u>

Functions in TypeScript work the same as JavaScript functions but with type annotations for parameters and return types.

<u>1. BASIC FUNCTION</u>

A simple function that returns a value or performs an operation.

Example:

```
function greet(name: string): string {

return `Hello, ${name}!`;

}

console.log(greet('Alice')); // Prints: Hello, Alice!
```

2. FUNCTION WITH OPTIONAL PARAMETERS:

You can define parameters as optional by appending ? to the parameter name.

Example:

```
function greetWithAge(name: string, age?: number): string {

if (age) {

return `Hello, ${name}! You are ${age} years old.`;

} else {

return `Hello, ${name}!`;

}

}
```

```
console.log(greetWithAge('Alice')); // Prints: Hello, Alice!

console.log(greetWithAge('Bob', 30)); // Prints: Hello, Bob! You are 30 years old.
```

3. FUNCTION TYPE

You can specify the function signature with explicit parameter types and return types.

Example:

```
let myAdd: (x: number, y: number) => number;

myAdd = (x, y) => x + y;

console.log(myAdd(2, 3)); // Prints: 5
```

4. RETURN TYPE

- The return type of a function is explicitly defined after the parameter list.
- TypeScript can also infer the return type in many cases.

Example:

```
function multiply(a: number, b: number): number {

return a * b;

}

console.log(multiply(5, 2)); // Prints: 10
```

OBJECTS :

- In TypeScript, objects are used to represent non-primitive data types.
- These objects consist of key-value pairs, where the key is a string and the value can be any valid type, including another object, a number, a string, etc.
- The object type is used to denote any value that is not a primitive type, such as a number, string, boolean, etc.

DEFINING OBJECTS

- You can define the types of properties within an object.
- This helps to enforce the structure and types of the properties that the object can have.

Example:

```
let user: { name: string, age: number } = {

name: 'Alice',

age: 30
};
```

- In this case, user is an object where the name is a string and the age is a number.
- TypeScript will throw an error if you try to assign values of incorrect types.

```
// This would throw an error

user = {

name: 123,  // Error: 'name' should be a string

age: '30'   // Error: 'age' should be a number

};
```

COMPLEX OBJECTS:

- TypeScript also allows for more complex object types.
- You can define objects where one or more properties may be functions or arrays, or even a combination of both.

Example:

```
let complexObject: { data: number[], transform: (input: number) => number } = {

data: [10, 20, 30],

transform: (input: number) => input * 2

};
```

- Here, complexObject has two properties:
- data, which is an array of numbers.
- transform, which is a function that takes a number as input and returns a number.

OPTIONAL OBJECT PROPERTIES:

- TypeScript allows you to define optional properties in objects using the ? syntax.

- This means that the property may or may not be present in the object.

Example:

```
let person: { name: string, age?: number } = {

name: 'John'

};

let anotherPerson: { name: string, age?: number } = {

name: 'Sarah',

age: 25

};
```

- In this case, the age property is optional. You can define an object with only the name property, and TypeScript will not complain.
- If the age property is provided, it must be a number.

ALIASES:

TypeScript allows you to create aliases for complex types, which helps in avoiding repetitive code and improving readability.

Example:

```
type Employee = { name: string, role: string, age: number };

let employee1: Employee = {

name: 'Peter',

role: 'Developer',

age: 28

};
```

- ❖ In this case, Employee is a type alias for objects that have a name, role, and age property.
- ❖ You can reuse the Employee type wherever needed.

UNION TYPES:

- ❖ TypeScript provides the ability to define a variable that can hold multiple types.
- ❖ This is called a union type. You can use the | operator to specify that a value can be one of several types.

Example:

```
let age: number | string = 25;  // Can be a number or a string

age = '25';  // This is valid

// age = true;  // Error: boolean is not allowed
```

- ❖ In this case, the variable age can be either a number or a string.
- ❖ However, it cannot hold any other type, such as a boolean.

UNION TYPES WITH OBJECTS:

- ❖ Union types can also be used within object properties.
- ❖ This allows the property to have more than one type of value.

Example:

```
let userProfile: { name: string, details: string | number } = {

name: 'Emily',

details: 30

};userProfile.details = 'Age 30';  // This is also valid
```

Here, the details property can be either a string or a number, and TypeScript will enforce that when assigning values to it.

INTERSECTION TYPES :

- ❖ Intersection types are a way to combine multiple types into one.
- ❖ They are useful when you want to represent an entity that shares the properties of several different types.
- ❖ This allows you to create complex structures by combining the capabilities of different types in a type-safe way.

What are Intersection Types?

- ❖ An intersection type is defined using the & operator, and it creates a new type that contains all the properties from the types involved.
- ❖ This is similar to combining features from multiple classes or interfaces into a single class.
- ❖ The resulting type will have the properties of both types, as long as the types do not conflict.
- ❖ For example, if you want to combine a Person type and a Loggable interface into a new LoggablePerson type,

you could write:

```
interface Loggable {

log(name: string, age: number): void;

}

interface Person {

name: string;

age: number;

isStark?: boolean;

}

type LoggablePerson = Loggable & Person;



const logPerson = (name: string, age: number) => {

console.log(`I am ${name}, and I am ${age} years old.`);

};



const jonSnow: LoggablePerson = {

name: "Jon Snow",

age: 23,

log: logPerson,

};

console.log(jonSnow.name); // Jon Snow
console.log(jonSnow.age);  // 23

jonSnow.log(jonSnow.name, jonSnow.age); // I am Jon Snow, and I am 23 years old.
```

In this example, the LoggablePerson type combines the properties of Person and Loggable, meaning jonSnow has the properties name, age, isStark, and a method log.

LIMITATIONS OF INTERSECTION TYPES

- Intersection types can only combine types that are structurally compatible.
- If two types have properties with conflicting structures, they cannot be combined.
- For instance, trying to intersect string and number types will result in an error because they are incompatible

```
let stringAndNumber: string & number = 5; // Error: Type 'number' is not assignable to type 'string'.
```

- This happens because TypeScript checks if types are compatible by comparing their properties.
- Since string and number don't share compatible properties, the intersection type fails.

USING INTERSECTION TYPES WITH CLASSES

- ❖ When working with classes, it’s better to use the extends keyword to create a subclass rather than using intersection types.
- ❖ Intersection types are ideal for combining interfaces or simpler types.

```
class Animal {

name: string;

constructor(name: string) {   this.name = name;       }}

interface Runner  {  run(): void; }

type RunningAnimal = Animal & Runner;

const cheetah: RunningAnimal = {

name: "Cheetah",

run: () => console.log("Running fast!"),

};

console.log(cheetah.name); // Cheetah

cheetah.run(); // Running fast!
```

In this case, RunningAnimal combines both the Animal class and the Runner interface, resulting in a new type that has both the name property and the run method.

TYPE GUARDS (CHECK)

- ❖ In TypeScript, you can use type guards to check the type of a variable at runtime.
- ❖ This is done using typeof for primitive types or instanceof for object types.

Example:

```
let value = "Hello, world!";
```

```
if (typeof value === "string") {

console.log("Value is a string");

}
```

Here, typeof ensures that the code block only runs if value is indeed a string.

THE NEVER TYPE

- The never type represents values that never occur.
- It is often used as the return type for functions that throw exceptions or that never complete.

Example:

```
function throwError(): never {

throw new Error("An error occurred!");  }
```

Since the function always throws an error, it never returns a value, making its return type never.

NULLABLE TYPES

- In TypeScript, you can explicitly define a variable that can either be null or another type, using union types.
- This helps represent cases where a value might be absent.

Example:

```
let canBeNull: null | number = 42;

canBeNull = null; // This is allowed.

Here, canBeNull can either be null or a number.
```

TYPE ASSERTIONS:

- Type assertions allow you to tell TypeScript to treat a value as a specific type when you know more about its type than TypeScript can infer.
- This is useful when working with any types.
- There are two syntaxes for type assertions:
- Angle bracket syntax (works in most situations):

```
let value: any = "Hello";

let length: number = (<string>value).length;

as syntax (recommended in JSX):

let value: any = "Hello";

let length: number = (value as string).length;
```

MODERN JAVASCRIPT FEATURES (ES6+) :

- TypeScript supports modern JavaScript features like arrow functions, template literals, destructuring, and more.
- These features work seamlessly with TypeScript’s type system.

ARROW FUNCTIONS WITH TYPES

Arrow functions are concise and can accept types for their parameters:

```
const greet = (name: string) => {

console.log(`Hello, ${name}`);

};

greet("Robert"); // Prints: Hello, Robert
```

DEFAULT PARAMETERS:

You can specify default values for function parameters:

```
const greet = (name: string = "Guest") => {

console.log(`Hello, ${name}`);

};

greet(); // Prints: Hello, Guest

greet("Alice"); // Prints: Hello, Alice

Spread Operator

The spread operator allows you to expand arrays or objects:

const numbers = [1, 2, 3];

console.log(Math.max(...numbers)); // Prints: 3

const newArray = [4, 5, ...numbers];

console.log(newArray); // Prints: [4, 5, 1, 2, 3]
```

DESTRUCTURING

Destructuring allows you to unpack values from arrays or objects:

```
const scores = [90, 85, 88];

const [firstScore, secondScore] = scores;

console.log(firstScore, secondScore); // 90 85

For objects:

const person = { name: "Alice", age: 25 };
const { name, age } = person;

console.log(name, age); // Alice 25
```

ANGULAR

- Angular is a popular web development platform developed and maintained by Google.
- Angular uses TypeScript as its main programming language.
- The Visual Studio Code editor supports TypeScript IntelliSense and code navigation out of the box, so you can do Angular development without installing any other extension.

HISTORY OF ANGULAR:

- Angular 2.0 was first announced at the ng-Europe conference on October 22-23, 2014.

It progressed through various stages:

- April 30, 2015: Angular 2 moved from Alpha to Developer Preview.
- December 2015: It reached the Beta stage.
- May 2016: The first release candidate was published.
- September 14, 2016: The final version of Angular 2 was officially released.

- Version 8 introduced a new compilation and rendering pipeline called Ivy, and Angular 9 made Ivy the default. With Angular 13, the older compiler, View Engine, was removed.

NAMING CLARIFICATION:

- To avoid confusion, the rewrite of AngularJS was named Angular 2, but the community later clarified that AngularJS refers to versions 1.X, while Angular (without "JS") refers to version 2 and later.

- Angular, also known as Angular 2+, is a TypeScript-based open-source framework for building single-page web applications (SPAs).

- Developed by Google and supported by a large community, Angular is a complete rewrite of AngularJS.

- It has become one of the most widely used frameworks, with over 1.7 million developers contributing to its ecosystem.

DIFFERENCES BETWEEN ANGULAR AND ANGULARJS :

- Angular is a framework used for building single-page applications (SPAs).
- It is a complete, open-source solution created by Google that includes tools for routing, forms management, HTTP handling, dependency injection, and more.
- Angular was rewritten from AngularJS (version 1.x) with a new architecture and many improvements, including support for TypeScript.
- Angular itself is written in TypeScript, but you can write Angular code using either TypeScript or JavaScript, though TypeScript is the preferred and recommended language due to its features like static typing, type checking, and intellisense support.

key differences between angular and angularjs:

- **Architecture:** Angular replaces AngularJS's "scope" and controllers with a component-based architecture.

- **Syntax:** Angular uses a new binding syntax with [ ] for property binding and ( ) for event binding.
- **TypeScript:** Unlike AngularJS, Angular emphasizes using TypeScript, which offers static typing, generics, and type annotations.

STARTING A NEW ANGULAR PROJECT:

Prerequisites:

- Before you can start developing with Angular, you need to have Node.js installed on your machine.
- Node.js comes with NPM (Node Package Manager), which is used to install the required Angular packages and dependencies.

Installing Angular CLI

- Angular provides a command-line interface (CLI) to streamline the development process.
- The Angular CLI helps in creating new projects, serving your app locally, building for production, adding new components, and more.

To install the Angular CLI, use either NPM or Yarn:

```
NPM:

        npm install -g @angular/cli

Yarn:

      yarn global add @angular/cli
```

Once installed, you can create a new Angular project with the following command:

```
        ng new my-app
```

Project Setup:

- When you run the above command, Angular will prompt you for some configuration options (e.g., adding Angular routing, choosing stylesheets like CSS or SCSS).
- After confirming your choices, Angular will set up a new project with the default configurations.

Running the Angular App:

To serve the app locally for development purposes:

```
ng serve
```

For production, you need to build the app with:

```
ng build –prod
```

- n an Angular project, the directory structure is designed to keep the project organized and maintainable.
- Each folder within an Angular project serves a specific purpose, helping developers efficiently structure their code, manage assets, and separate concerns.
- Below is a breakdown of the typical folders you will encounter in an Angular project and what they are used for.

1. Root Folder (Project Directory)

When you create a new Angular project using the Angular CLI, you'll see the following default structure at the root level:

```
my-app/

├── node_modules/

├── src/

├── angular.json

├── package.json
├── tsconfig.json

├── tsconfig.app.json

├── tslint.json

├── e2e/

└── .gitignore
```

2. src/ Folder

The src/ folder (short for "source") is where most of your application’s code resides. This includes the core application logic, components, templates, and styles. When you build the application, Angular will bundle everything in this folder into the final output.

Inside the src/ folder, you'll typically find the following subfolders:

2.1. src/app/ Folder

This is the most important folder in an Angular project as it contains the main code for your application, including:

Components: Reusable UI elements of your app. Each component typically includes a .ts file (TypeScript), a .html file (template), a .css/.scss file (styling), and a .spec.ts file (for testing).

- Example: src/app/my-component/my-component.component.ts

Services: Services are classes that encapsulate business logic, data retrieval, or any reusable functionality that can be shared across components.

- Example: src/app/services/data.service.ts

Modules: Angular modules are used to group related components, services, pipes, etc., into functional units. The root module of your Angular application is called AppModule, located in src/app/app.module.ts.

- Example: src/app/app.module.ts

Routing: If your application has multiple views or pages, the routing module (app-routing.module.ts) handles navigation and route management.

- ❖ Example: src/app/app-routing.module.ts

2.2. src/assets/ Folder

The assets/ folder holds static files such as images, stylesheets, icons, fonts, and any other non-code resources that need to be served by the application. It is commonly used to store:

- ➢ Images (e.g., .jpg, .png, .svg)
- ➢ Fonts (e.g., .woff, .ttf)
- ➢ Stylesheets (e.g., .css, .scss, .less)

External libraries or files

Example:

```
src/assets/

├── images/

├── icons/

├── styles/

└── data/
```

These files are publicly accessible and can be referenced in templates or styles.

2.3. src/environments/ Folder

- ❖ This folder contains environment configuration files that are used to define different settings for different environments (e.g., development, production).
- ➢ environment.ts: The default environment settings for development.
- ➢ environment.prod.ts: The production environment settings.
- ❖ These files help configure settings like API URLs, logging levels, and feature flags based on the environment the application is running in.

- Angular uses the fileReplacements option in the angular.json file to switch between these files during builds.

Example:

```
src/environments/
├── environment.ts      // Development environment settings
└── environment.prod.ts  // Production environment settings
```

2.4. src/styles/ Folder

- If you're using global styles across your app, the styles/ folder is where they go.
- This folder can contain CSS, SCSS, or any other stylesheet you use globally in the app.
- styles.css (or styles.scss): The main global stylesheet file for your project.

Example:

```
src/styles/
└── styles.css
```

2.5. src/favicon.ico

- This file represents the favicon (the small icon displayed in the browser tab) for the application.
- It's a default file placed in the root src/ folder.

2.6. src/main.ts

- This is the main entry point of the Angular application.
- It is responsible for bootstrapping (starting) the Angular application by calling platformBrowserDynamic().bootstrapModule(AppModule).

2.7. src/polyfills.ts

- This file is used to include polyfills that enable the app to run on older browsers.

- ❖ It ensures compatibility with browsers that don't support some modern JavaScript features natively (e.g., older versions of Internet Explorer).

3. e2e/ Folder (End-to-End Tests)

- ❖ The e2e/ folder contains the configuration and code for end-to-end testing of the Angular application.
- ❖ These tests simulate user interactions with the application, ensuring the entire system works as expected from a user's perspective.
- ❖ Protractor is typically used for e2e testing in Angular, and the configuration files and test scripts are stored in this folder.

Example:

```
e2e/

├── src/

│   └── app.e2e-spec.ts   // E2E test cases

└── protractor.conf.js    // Configuration file for Protractor
```

4. Configuration Files at the Root Level

4.1. angular.json

This is the main configuration file for an Angular project. It defines settings like:

Project structure

- ❖ Build and development settings (e.g., file replacements, build optimizations)
- ❖ Architect configurations (e.g., build, test, serve)

4.2. package.json

- ❖ This file lists all the dependencies (both development and production), scripts, and metadata for the project.

- It is where you define npm scripts to run Angular CLI commands like ng serve, ng build, etc.

4.3. tsconfig.json

- This TypeScript configuration file defines how TypeScript files should be compiled.
- It includes options like paths, module resolution, and target ECMAScript versions.

4.4. tsconfig.app.json

- This file is an extension of tsconfig.json and is specifically used for configuring TypeScript compilation for the application code.

4.5. tslint.json

- This configuration file is used to define linting rules for TypeScript.
- It helps ensure that your code adheres to style guidelines and best practices.

4.6. .gitignore

- This file tells Git which files and directories to ignore when committing to a repository.
- It typically includes directories like node_modules, build artifacts, and IDE-specific files.

5. node_modules/ Folder

- This folder is created when you install dependencies using npm install.
- It contains all the libraries and packages that the Angular project depends on.
- You should not manually modify files in node_modules/, as they are automatically managed by npm.

6. Build and Output Files

dist/ folder:

- After running the build command (ng build), Angular compiles the project and stores the final, optimized production files (HTML, JavaScript, CSS) in the dist/ directory.
- This is the folder that gets deployed to production.

COMPONENTS IN ANGULAR:

- The building blocks of an application
- Components are the foundational building blocks of an Angular application.
- They essentially teach browsers new HTML tags with custom behavior.
- Components are typically created using the Angular CLI, which provides commands for both standard and shorthand usage.

Creating a Component with the Angular CLI:

You can use the following commands to generate a new component:

Command:

```
ng generate component MyNewComponent

        or

ng g c MyNewComponent
```

When you run either command, Angular will create the component files in a directory named after the component.

The following files will be generated:

my-new-component.component.html :

- The template file containing the HTML structure displayed when the component is rendered.

my-new-component.component.css:

- The CSS file encapsulating the component's styles.

my-new-component.component.ts:

- The Typescript file containing the component's logic and behavior.

my-new-component.component.spec.ts:

❖ A test file for validating the behavior and output of the component.

Additionally, the component's class is automatically decorated with the @Component decorator and registered with the nearest Angular module.

Common CLI Options for Generating Components :

Here are some helpful options you can use with the Angular CLI when creating components:

--dry-run (-d) ng g c MyNewComponent –d :

Simulates the command without creating files. Useful for previewing changes.

--export ng g c MyNewComponent –export:

Exports the component, making it available for use in other modules.

--force (-f) ng g c MyNewComponent –f:

Overwrites existing files if the component already exists.

--help ng g c –help:

Lists all available options for the command.

--prefix (-p) ng g c MyNewComponent -p=custom:

Sets a custom prefix for the component's HTML selector.

--skip-tests ng g c MyNewComponent --skip-tests:

Skips generating the .spec.ts test file for the component.

--style ng g c MyNewComponent --style=sass :

Specifies the style file's format (e.g., SCSS, SASS, or NONE).

LIFECYCLE HOOKS IN ANGULAR:

❖ Angular components go through a series of lifecycle stages, emitting events at each stage.

❖ Developers can hook into these events by implementing specific methods in the component class.

Below is an overview of the lifecycle hooks:

ngOnChanges

Invoked after the value of an input property changes.

ngOnInit

Called once the component is initialized and input bindings are ready.

ngDoCheck

Enables custom change detection logic.

ngAfterContentInit

Runs after the projected content within a component is initialized.

ngAfterContentChecked

Runs after every check of the projected content.

ngAfterViewInit

Invoked after the component's view and child views are fully initialized.

ngAfterViewChecked

Runs after every check of the component's view and its child views.

ngOnDestroy

Called before the component is removed or destroyed. Useful for cleanup tasks like unsubscribing from observables.

Examples of Lifecycle Hook Implementations

ngOnInit

- This is commonly used to initialize component data or perform setup tasks like fetching data.

```
ngOnInit(): void {

  console.log('Component Initialized');

  this.fetchUserData();

}
```

ngOnChanges

- This runs whenever an input property value changes.
- It is especially useful when a parent component passes new data to a child component.

```
ngOnChanges(changes: SimpleChanges): void {

  if (changes['userName']) {

    console.log('userName changed:', changes['userName'].currentValue);

  }

}
```

ngOnDestroy

- This is used for cleanup, such as unsubscribing from observables or clearing timers.

```
ngOnDestroy(): void {

  console.log('Component is being destroyed');

  this.subscription.unsubscribe();
}
```

- ❖ In summary, Angular components combine HTML, CSS, and TypeScript to create modular, reusable UI elements.
- ❖ By leveraging the Angular CLI and lifecycle hooks, developers can efficiently build and manage the application's dynamic behavior and data flow.
- ❖

ANGULAR SERVICES: OUTSOURCING LOGIC AND DATA

- ❖ In Angular, services are objects designed for outsourcing reusable logic and data that can be shared across multiple components or modules.
- ❖ Services enable cleaner, modular, and more maintainable code, making them essential for medium and large-scale applications.
- ❖ For medium-sized apps, services can even act as a lightweight alternative to state management libraries.

Creating a Service with Angular CLI

- ❖ Services are typically created using Angular CLI. Here’s how:

Standard Command:

```
ng generate service MyService

          or

ng g s MyService
```

When you create a service, Angular generates two files:

- ❖ **my-service.service.ts:** The file containing the service logic.

- ❖ **my-service.service.spec.ts:** A test file for verifying the functionality of the service.

INJECTING SERVICES IN ANGULAR:

- Services are not standalone; they are meant to be injected into other parts of the app, such as components, directives, or other services.

- To make a class a service and injectable, you need to follow these two steps:

Step 1: Add the @Injectable() Decorator

The @Injectable() decorator marks a class as a service and makes it available for dependency injection.

```
import { Injectable } from '@angular/core';

@Injectable()

export class MyService {

  constructor() { }

}
```

Step 2: Specify the Injection Scope

- After making the class injectable, you need to tell Angular where the service should be available.
- Injectable at the Root Level The most common method is to provide the service in the root injector. This ensures the service is a singleton and accessible throughout the application.

```
@Injectable({

  providedIn: 'root' // The service is provided in the root injector

})

export class MyService {
  constructor() { }

}
```

Advantages:

- Automatically tree-shakeable (removed from the bundle if unused).
- No need to register the service in a module’s providers array.
- Injectable at the Module Level If you want to restrict the service's scope to a specific module, you can provide it in the module's providers array.

```
import { NgModule } from '@angular/core';

import { MyService } from './my-service.service';

@NgModule({

  declarations: [],

  imports: [],

  providers: [MyService], // The service is available only in this module

  bootstrap: []

})

export class MyModule { }
```

Use Case:

- When a service is specific to a feature module and not needed globally.
- Injectable at the Component Level You can also provide a service at the component level by adding it to the providers array in the component’s decorator.
- This creates a new instance of the service for that component and its children.

```
import { Component } from '@angular/core';
import { MyService } from './my-service.service';

@Component({

  selector: 'app-my-component',

  templateUrl: './my-component.component.html',

  styleUrls: ['./my-component.component.css'],

  providers: [MyService] // New instance of the service for this component

})

export class MyComponent {

  constructor(private myService: MyService) { }
```

Use Case:

When a service should have a separate instance for each component (e.g., for component-specific state management).

MODULES :

- Angular enhances JavaScript's modularity with its own module system.

- Classes decorated with the @NgModule() decorator can register components, services, directives, and pipes.

The following options can be added to a module:

declarations - List of components, directives, and pipes that belong to this module.

imports - List of modules to import into this module. Everything from the imported modules is available to declarations of this module.

exports - List of components, directives, and pipes visible to modules that import this module.

providers - List of dependency injection providers visible both to the contents of this module and to importers of this module.

bootstrap - List of components to bootstrap when this module is bootstrapped.

Here's an example of a module called AppModule.

```
import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

@NgModule({

  declarations: [  AppComponent ],

  imports: [  BrowserModule,  AppRoutingModule ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule {

 }
```

ANGULAR DIRECTIVES:

- Directives are custom attributes that can be applied to elements and components to modify their behavior.
- There are two types of directives:
    - attribute directives
    - structural directives

ATTRIBUTE DIRECTIVES:

- ❖ An attribute directive is a directive that changes the appearance or behavior of an element, component, or another directive.
- ❖ Angular exports the following attribute directives:

NgClass

Adds and removes a set of CSS classes.

```
<!-- toggle the "special" class on/off with a property -->

<div [ngClass]="isSpecial ? 'special' : ''">This div is special</div>
```

NgStyle

Adds and removes a set of HTML styles.

```
<div [ngStyle]="{

  'font-weight': 2 + 2 === 4 ? 'bold'   : 'normal',

}">

  This div is initially bold.

</div>
```

NgModel

Adds two-way data binding to an HTML form element.

Firstly, this directive requires the FormsModule to be added to the @NgModule() directive.

```
import { FormsModule } from '@angular/forms'; // <--- JavaScript import from Angular

/* . . . */
```

```
@NgModule({

 /* . . . */

 imports: [

   BrowserModule,

   FormsModule // <--- import into the NgModule

 ],

 /* . . . */

})

export class AppModule { }
```

Secondly, we can bind the [(ngModel)] directive on an HTML <form> element and set it equal to the property.

```
<label for="example-ngModel">[(ngModel)]:</label>

<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

The `NgModel` directive has more customizable options that can be [found here](https://angular.io/guide/built-in-directives#displaying-and-updating-properties-with-ngmodel).

STRUCTURAL DIRECTIVES:

- Structural directives are directives that change the DOM layout by adding and removing DOM elements.
- Here are the most common structural directives in Angular:

NgIf

A directive that will conditionally create or remove elements from the template. If the value of the NgIf directive evaluates to false, Angular removes the element.

```
<p *ngIf="isActive">Hello World!</p>
```

NgFor

Loops through an element in a list/array.

```
<div *ngFor="let item of items">{{item.name}}</div>
```

NgSwitch

- ❖ An alternative directive for conditionally rendering elements.
- ❖ This directive acts very similarly to the JavaScript switch statement. There are three directives at our disposal:

- ❖ NgSwitch — A structural directive that should be assigned the value that should be matched against a series of conditions.

- ❖ NgSwitchCase — A structural directive that stores a possible value that will be matched against the NgSwitch directive.

- ❖ NgSwitchDefault — A structural directive that executes when the expression doesn't match with any defined values.

```
<ul [ngSwitch]="food">

  <li *ngSwitchCase="'Burger'">Burger</li>

  <li *ngSwitchCase="'Pizza'">Pizza</li>

  <li *ngSwitchCase="'Spaghetti'">Spaghetti</li>

  <li *ngSwitchDefault>French Fries</li>
```

```
</ul>
```

<u>CUSTOM DIRECTIVES</u>

- custom directive in Angular, which dynamically changes the background color of an element when the mouse enters or leaves it.

Step 1: Generate a Directive

Use the Angular CLI to create the directive:

```
ng generate directive HoverHighlight
```

Step 2: Implement the Custom Directive

Open the hover-highlight.directive.ts file and implement the directive logic:

```
import { Directive, ElementRef, HostListener, Input, Renderer2 } from '@angular/core';

@Directive({

  selector: '[appHoverHighlight]' // Directive applied as an attribute

})

export class HoverHighlightDirective {

  @Input('appHoverHighlight') highlightColor: string = 'yellow'; // Input property for dynamic color

  constructor(private el: ElementRef, private renderer: Renderer2) {}

  // Listen to mouseenter event

  @HostListener('mouseenter') onMouseEnter() {

    this.setBackgroundColor(this.highlightColor || 'yellow'); // Apply highlight color

  }
  // Listen to mouseleave event

  @HostListener('mouseleave') onMouseLeave() {

    this.setBackgroundColor(''); // Reset background color

  }

  private setBackgroundColor(color: string) {

    this.renderer.setStyle(this.el.nativeElement, 'background-color', color);

  }

}
```

KEY CONCEPTS IN THE DIRECTIVE

@Directive Decorator:

- ❖ Defines the directive with the selector, which is used as an attribute in templates.

@Input Decorator:

- ❖ Allows the directive to accept input properties for customization (e.g., setting a dynamic highlight color).

@HostListener Decorator:

- ❖ Listens to events (e.g., mouseenter, mouseleave) on the host element and triggers specified methods.

ElementRef and Renderer2:

- ❖ ElementRef gives access to the DOM element where the directive is applied.
- ❖ Renderer2 provides a safe way to modify the DOM (e.g., setting styles).

Step 3: Use the Directive in a Component

```
HTML Template (app.component.html):
```

```
<h3>Hover over the elements below:</h3>

<div appHoverHighlight="lightblue">Hover to see the light blue highlight!</div>

<br>

<div appHoverHighlight="lightgreen">Hover to see the light green highlight!</div>

<br>

<div appHoverHighlight>Hover to see the default yellow highlight!</div>
```

Step 4: Add the Directive to a Module

Make sure the directive is declared in the AppModule or the relevant feature module:

app.module.ts:

```
import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { HoverHighlightDirective } from './hover-highlight.directive'; // Import the directive

@NgModule({

  declarations: [

    AppComponent,

    HoverHighlightDirective // Declare the directive

  ],

  imports: [BrowserModule],

  providers: [],
  bootstrap: [AppComponent]

})

export class AppModule {}
```

Step 5: Run the Application

Run the application using the Angular CLI:

```
ng serve
```

Expected Behavior:

- When you hover over the first <div>, its background changes to light blue.
- When you hover over the second <div>, its background changes to light green.
- When you hover over the third <div>, its background changes to the default yellow.
- The background color resets when you move the mouse away.

PIPES :

- Pipes in Angular are used to transform displayed content in templates without altering the actual data.
- They are especially handy for formatting or transforming data on the fly.
- Pipes are applied in templates using the | symbol.

Here's a simple example:

```
{{ 'Angular Pipes' | lowercase }}
```

This will transform the text to lowercase: angular pipes.

Built-in Pipes in Angular:

- Angular provides several built-in pipes for common transformations.
- Below are some of the most commonly used pipes with updated examples:

DatePipe

- Formats a date value according to locale-specific rules.
- You can specify formats like 'short', 'medium', 'long', or even custom formats.

Example:

```
{{ '2024-11-28T15:30:00' | date:'fullDate' }}
```

Output: Thursday, November 28, 2024

UpperCasePipe

- Transforms a string to all uppercase letters.

Example:

```
{{ 'learning angular' | uppercase }}
```

Output: LEARNING ANGULAR

LowerCasePipe

- Transforms a string to all lowercase letters.

Example:

```
{{ 'LEARNING Angular' | lowercase }}
```

Output: learning angular

CurrencyPipe

- Formats a number as a currency string.
- You can specify the currency type and formatting options.

Example:

```
{{ 1234.56 | currency:'EUR' }}
```

Output: €1,234.56

DecimalPipe

- Formats a number with decimal points.
- You can specify the minimum and maximum number of digits before and after the decimal.

Example:

```
{{ 7.56789 | number:'1.1-3' }}
```

Output: 7.568

Explanation: At least 1 digit before the decimal, 1-3 digits after the decimal.

PercentPipe

- Transforms a number into a percentage string.

Example:

```
{{ 0.4567 | percent:'1.0-2' }}
```

Output: 45.67%

Using Multiple Pipes

Pipes can also be chained together for complex transformations:

Example:

```
{{ 'Angular Pipes' | uppercase | slice:0:7 }}
```

Output: ANGULAR

Explanation: The string is transformed to uppercase and then sliced to display only the first 7 characters.

FORMS :

Angular provides two approaches to handling forms:

- ❖ Template-driven forms
- ❖ Reactive forms.

Both approaches offer powerful features for gathering user input, performing validations, and managing form state, but they differ in their structure and implementation style.

1. TEMPLATE-DRIVEN FORMS:

- ❖ Template-driven forms are forms where most of the logic is handled in the HTML template.
- ❖ They are suitable for simpler forms and are declarative in nature.

Key Features:

Two-way Data Binding: Template-driven forms use ngModel to bind form elements to properties in the component.

FormsModule: You need to import the FormsModule to use template-driven forms.

Automatic Validation: Validation is done through the template by using Angular's built-in directives.

Example:

```
<!-- app.component.html -->

<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">

  <label for="name">Name:</label>

  <input type="text" id="name" name="name" ngModel required>

  <label for="email">Email:</label>

  <input type="email" id="email" name="email" ngModel email>

  <button type="submit" [disabled]="myForm.invalid">Submit</button>
```

```
</form>
```

// app.component.ts

```
import { Component } from '@angular/core';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  onSubmit(form: any): void {

    console.log('Form Submitted', form);

  }

}
```

In this example:

- ngModel binds the form inputs to the component's properties.
- The #myForm="ngForm" creates a reference to the form, which can be used to track its validity.
- The form is submitted using the ngSubmit event, and the form is disabled when it is invalid.

Advantages of Template-driven Forms:

- Simple and easy to implement.
- Best for simple forms with minimal logic.

- ❖ Disadvantages of Template-driven Forms:
- ❖ Less flexibility and scalability for complex forms.
- ❖ Harder to manage large forms with dynamic behavior.

2. REACTIVE FORMS:

Reactive forms provide a more programmatic approach to form creation, allowing the form model to be created and controlled within the component. This approach is more flexible and scalable, suitable for complex forms.

Key Features:

Reactive Programming: Reactive forms are based on the FormControl and FormGroup classes.

ReactiveFormsModule: You need to import the ReactiveFormsModule to use reactive forms.

Manual Form Control: Form fields are explicitly defined in the component class using FormControl and FormGroup.

Form Validation: You can define synchronous and asynchronous validators in the component.

Example:

```
// app.component.ts

import { Component } from '@angular/core';

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})
```

```
export class AppComponent {

  myForm: FormGroup;

  constructor(private fb: FormBuilder) {

    this.myForm = this.fb.group({

      name: ['', [Validators.required, Validators.minLength(3)]],

      email: ['', [Validators.required, Validators.email]]

    });

  }

  onSubmit(): void {

    if (this.myForm.valid) {

      console.log('Form Submitted', this.myForm.value);

    }

  }

}
```

```
<!-- app.component.html -->

<form [formGroup]="myForm" (ngSubmit)="onSubmit()">

  <label for="name">Name:</label>

  <input type="text" id="name" formControlName="name">
  <label for="email">Email:</label>

  <input type="email" id="email" formControlName="email">

  <button type="submit" [disabled]="myForm.invalid">Submit</button>

</form>
```

In this example:

- The form is created using FormBuilder and is initialized with validators.
- The formGroup directive binds the form to the template.
- Each form control (name, email) is bound using formControlName.

Advantages of Reactive Forms:

- More control over form state and validation logic.
- More flexible and suitable for complex forms.
- Easier to test and manage dynamic forms.

Disadvantages of Reactive Forms:

- More verbose and complex setup compared to template-driven forms.
- Requires more boilerplate code for simpler forms.

3. Form Validation

Form validation in Angular can be done either declaratively (in templates) or programmatically (in the component). Both template-driven and reactive forms support built-in validation.

Built-in Validators:

Angular provides several built-in validators for common use cases:

- required: Ensures the field is not empty.
- minLength(length): Ensures the input has a minimum length.
- maxLength(length): Ensures the input has a maximum length.
- email: Ensures the input is a valid email.

- pattern(regex): Ensures the input matches a specified pattern.

Example of Built-in Validation (Reactive Forms):

```
this.myForm = this.fb.group({

  name: ['', [Validators.required, Validators.minLength(3)]],

  email: ['', [Validators.required, Validators.email]],

});
```

You can check the validation status using valid, invalid, pending, or touched properties:

```
<div *ngIf="myForm.controls['name'].invalid && myForm.controls['name'].touched">

  Name is required and must be at least 3 characters long.

</div>
```

4. FormControl and FormGroup

FormControl: Represents an individual form control (like an input field). It tracks the value and validation status of the input.

```
nameControl = new FormControl('', [Validators.required, Validators.minLength(3)]);
```

FormGroup: A group of related FormControl instances. It tracks the validity and value of a group of controls.

```
this.myForm = new FormGroup({

  name: this.nameControl,

  email: new FormControl('', [Validators.required, Validators.email]),

});
```

You can access individual controls using this.myForm.get('controlName') and apply validations or retrieve values.

5. Custom Validators

Custom validators allow you to implement your own validation logic beyond the built-in validators.

Example of a Custom Validator:

```
import { AbstractControl, ValidationErrors, ValidatorFn } from '@angular/forms';

export function forbiddenNameValidator(forbiddenName: string): ValidatorFn {

  return (control: AbstractControl): ValidationErrors | null => {

    const forbidden = control.value && control.value.includes(forbiddenName);

    return forbidden ? { 'forbiddenName': { value: control.value } } : null;

  };

}

You can apply the custom validator as part of the form control:

this.myForm = this.fb.group({

  name: ['', [Validators.required, forbiddenNameValidator('admin')]],

});
```

In this example, the custom validator checks if the input contains the forbidden word "admin".

Async Custom Validators:

For asynchronous validation (like checking if an email already exists), you can create an async validator using observables.

```
import { Observable, of } from 'rxjs';

import { debounceTime, map } from 'rxjs/operators';

export function emailAsyncValidator(control: AbstractControl): Observable<ValidationErrors |
null> {

  const forbiddenEmails = ['test@example.com', 'admin@example.com'];

  return of(forbiddenEmails.includes(control.value) ? { 'forbiddenEmail': { value: control.value }
} : null)

    .pipe(debounceTime(500));

}
```

DECORATORS IN ANGULAR :

- ❖ Decorators in Angular are special functions used to enhance classes, methods, properties, or parameters.
- ❖ They provide metadata that Angular uses to understand and configure the behavior of the class or element to which they are applied.
- ❖ Angular heavily relies on decorators for its core features, such as components, services, and modules. Decorators are a TypeScript feature and are prefixed with an @ symbol.

Types of Angular Decorators

1. Class Decorators

- ❖ Class decorators are used to define metadata for classes.

- ❖ These are the most common decorators in Angular, such as @Component, @Directive, and @NgModule.

@Component - Used to define a component and associate its metadata.

Example:

```
import { Component } from '@angular/core';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'Angular Decorators';

}
```

@NgModule Used to define a module and associate its metadata.

Example:

```
import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule({

  declarations: [AppComponent],

  imports: [BrowserModule],

  bootstrap: [AppComponent]

})

export class AppModule {}
```

2. Property Decorators

- ❖ Property decorators are used to add metadata to class properties.

- ❖ Commonly used property decorators include @Input() and @Output().

@Input Allows data to be passed from a parent component to a child component.

Example:

```
import { Component, Input } from '@angular/core';

@Component({

  selector: 'app-child',

  template: `<p>{{ data }}</p>`

})

export class ChildComponent {

  @Input() data!: string;

}
```

@Output Allows a child component to emit events to the parent component.

Example:

```
import { Component, Output, EventEmitter } from '@angular/core';

@Component({

  selector: 'app-child',
  template: `<button (click)="sendData()">Send Data</button>`
})
export class ChildComponent {
  @Output() dataEmitter = new EventEmitter<string>();
  sendData() {
    this.dataEmitter.emit('Hello from Child!');
  }     }
```

3. Method Decorators

- Method decorators are used to add metadata or behavior to methods of a class.
- An example is @HostListener.

@HostListener Binds a DOM event to a method.

Example:

```
import { Directive, HostListener } from '@angular/core';
@Directive({            selector: '[appHover]'       })
export class HoverDirective {
  @HostListener('mouseenter') onMouseEnter() {
    console.log('Mouse entered!'); }
  @HostListener('mouseleave') onMouseLeave() {
    console.log('Mouse left!');
  }
}
```

4. Parameter Decorators

- Parameter decorators are used to inject dependencies into a class constructor.
- The most common is @Inject.

@Inject Specifies a dependency to be injected.

Example:

```
import { Component, Inject } from '@angular/core';

import { DOCUMENT } from '@angular/common';

@Component({

  selector: 'app-root',

  template: `<p>Check console for injected document object</p>`

})

export class AppComponent {

  constructor(@Inject(DOCUMENT) private document: Document) {

    console.log(this.document);

  }

}
```

5. Accessor Decorators

- Accessor decorators are used to intercept access to a property's getter or setter.
- These are less common in Angular.

Example:

```
function LogAccess(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

  const originalGet = descriptor.get!;

  descriptor.get = function () {

    console.log(`Getting value of ${propertyKey}`);

    return originalGet.apply(this);

  };

}

class Example {

  private _value: string = 'Angular';

  @LogAccess

  get value() {

    return this._value;

  }}const obj = new Example();

console.log(obj.value); // Logs: Getting value of value
```

<u>ROUTING IN ANGULAR</u>

- ❖ Routing in Angular enables navigation between different views or components based on the user's interaction with the application.

- It is a core feature of Angular, allowing developers to build single-page applications (SPAs) where views are rendered dynamically without requiring page reloads.

1. Overview of Angular Router

- The Angular Router is a module in Angular that allows you to define routes for navigating between views.
- It maps browser URLs to components and ensures the correct component is displayed based on the current route.
- To enable routing in an Angular application, you need to import the RouterModule from @angular/router and configure the routes.

2. Setting up Routes

- In Angular, routes are defined in the RouterModule by specifying an array of route objects.
- Each object defines a path and a component to load when the path is matched.

Steps to set up routing:

- Import RouterModule and Routes from @angular/router.
- Define routes in a routing module (often in a dedicated file like app-routing.module.ts).
- Configure routes using RouterModule.forRoot() in the root module (AppModule).

Example:

```
// app-routing.module.ts

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

import { HomeComponent } from './home/home.component';

import { AboutComponent } from './about/about.component';

const routes: Routes = [
  { path: '', component: HomeComponent }, // Default route

  { path: 'about', component: AboutComponent },

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule {}
```

In this example:

- ❖ The home page is mapped to the empty path (''), which serves as the default route.
- ❖ The about route maps to the AboutComponent.

3. Route Guards (e.g., CanActivate, CanDeactivate)

- ❖ Route guards are special services in Angular that allow you to control access to routes.
- ❖ They act as protection mechanisms and are useful for scenarios like authentication, confirming navigation, or preventing unsaved changes from being discarded.

CanActivate: Prevents navigation to a route based on specific conditions. For example, you can check if the user is logged in before allowing access to a certain route.

CanDeactivate: Prevents leaving a route based on conditions, such as confirming if a user has unsaved changes.

Example of CanActivate Guard:

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from
'@angular/router';

import { AuthService } from './auth.service';

@Injectable({

  providedIn: 'root',

})

export class AuthGuard implements CanActivate {

  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {

    if (this.authService.isAuthenticated()) {

      return true;

 } else {

      this.router.navigate(['/login']);

      return false;

    }

  }

}
```

In this example, the AuthGuard checks if the user is authenticated before allowing access to a route.

Example of CanDeactivate Guard:

```
import { Injectable } from '@angular/core';
import { CanDeactivate } from '@angular/router';

import { Observable } from 'rxjs';

import { Component } from '@angular/core';

export interface CanComponentDeactivate {

  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;

}

@Injectable({

  providedIn: 'root',

})

export class UnsavedChangesGuard implements CanDeactivate<CanComponentDeactivate> {

  canDeactivate(component: CanComponentDeactivate): Observable<boolean> | Promise<boolean> | boolean {

    return component.canDeactivate ? component.canDeactivate() : true;

  }

}
```

This guard is useful when a user is trying to navigate away from a page with unsaved changes. If the component implements the CanComponentDeactivate interface, it can return a confirmation prompt before leaving.

4. Nested Routes and Child Routes

- Nested routes (also known as child routes) allow you to create routes inside other routes.
- This is useful for creating hierarchical UI structures.

- ❖ You can define nested routes by placing them inside the children array of a parent route.

Example:

```
const routes: Routes = [
  {
    path: 'profile',
    component: ProfileComponent,
    children: [
      { path: 'details', component: ProfileDetailsComponent },
      { path: 'settings', component: ProfileSettingsComponent },
    ],
  },
];
```

- ❖ In this example, the ProfileComponent is the parent route, and it has two child routes: details and settings.
- ❖ The child routes will be displayed in the <router-outlet> inside the ProfileComponent.

Template Example:

```
<!-- profile.component.html -->
<h2>Profile</h2>
<router-outlet></router-outlet>
```

Here, the <router-outlet> will be used to display the child routes (ProfileDetailsComponent and ProfileSettingsComponent) based on the active route.

HTTP AND OBSERVABLES :

- Angular provides a powerful HttpClient module to interact with remote servers via HTTP requests.
- This module uses Observables (from RxJS, Reactive Extensions for JavaScript) to handle asynchronous operations like HTTP requests, making it easier to manage response data, handle errors, and chain multiple HTTP operations.

1. HttpClient Module

- The HttpClient module is part of Angular's HTTP package and provides a simplified API for making HTTP requests.

- It uses Observables for handling asynchronous responses, which allows for more powerful composition and manipulation of HTTP responses.

Key Features:

Type Safety: You can define the expected response type, making it easier to handle data.

Rich APIs: Supports methods for various HTTP operations (GET, POST, PUT, DELETE, etc.).

RxJS Support: Returns RxJS Observable objects that can be piped, mapped, and subscribed to, enabling reactive programming.

To use HttpClient, you need to import the HttpClientModule in your Angular module (AppModule).

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({

  imports: [HttpClientModule],

})

export class AppModule {}
```

Example of Injecting HttpClient into a Service:

```
import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root',

})

export class MyService {

  constructor(private http: HttpClient) {}

  // Example of a GET request

  getData(): Observable<any> {

    return this.http.get('https://api.example.com/data');

  }

}
```

In this example:

- ❖ The HttpClient is injected into the service.
- ❖ The getData() method sends a GET request to fetch data from the specified URL and returns an Observable that will emit the response data.

2. Making HTTP Requests

Angular's HttpClient provides methods to make different types of HTTP requests:

- **GET:** Retrieves data from the server.
- **POST:** Sends data to the server to create or update resources.
- **PUT:** Updates an existing resource.
- **DELETE:** Deletes a resource.
- **PATCH:** Partially updates a resource.

Example: Making a GET Request

```
this.http.get('https://api.example.com/data')

  .subscribe(response => {

    console.log('Data received:', response);

  });
```

Example: Making a POST Request

```
const newData = { name: 'John', age: 30 };

this.http.post('https://api.example.com/data', newData)

  .subscribe(response => {

    console.log('Data posted successfully', response);

  });
```

In both cases, the HTTP methods return an Observable which you can subscribe to, handle the response data, or catch errors.

3. Observables and RxJS

- ❖ In Angular, HTTP requests return Observables—a key concept in RxJS. Observables represent a stream of data that can emit multiple values over time, and they are asynchronous by nature.
- ❖ RxJS provides operators like map, catchError, mergeMap, etc., to manipulate, filter, and handle data streams.
- ❖ Example: Using map to Transform Response Data

```
import { map } from 'rxjs/operators';

this.http.get('https://api.example.com/data')

  .pipe(

    map((response: any) => response.items) // Extracting the 'items' from the response

  )

.subscribe(items => {

    console.log('Items:', items);

  });
```

Example: Using catchError to Handle Errors

```
import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

this.http.get('https://api.example.com/data')

  .pipe(

    catchError(error => {

      console.error('Error occurred:', error);

      return throwError(error); // Propagate error
    })

  )

 .subscribe();
```

In this example:

- The catchError operator is used to handle errors from the HTTP request, ensuring that your application can gracefully handle failures.
- map is used to transform the response data before subscribing to it.

4. Handling HTTP Responses and Errors

- Handling responses and errors effectively is a critical part of working with HTTP requests.
- The HttpClient makes it easy to deal with HTTP responses by returning an Observable that emits the response.
- This allows developers to react to different scenarios like success, failure, or retries.

Successful Response Handling

When an HTTP request is successful, the Observable emits the response data. You can access this data in the subscribe() method's callback function.

```
this.http.get('https://api.example.com/data')

  .subscribe(response => {

    console.log('Received data:', response);

  });
```

Error Handling

If the request fails (e.g., due to a 404 or network issue), you can catch the error using the catchError operator and handle it appropriately (e.g., displaying an error message to the user).

```
import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

this.http.get('https://api.example.com/data')

  .pipe(

    catchError(error => {

      console.error('Error:', error);

      return throwError('Something went wrong');

    })

  )

  .subscribe(

    data => {

      console.log('Data:', data);

    },

    error => {

      console.error('Handled Error:', error);

    }

  );
```

In this example:

- ❖ If an error occurs (e.g., network failure or 404 error), it is caught and handled by the catchError operator.
- ❖ The throwError() function re-throws the error or provides a fallback value.

5. HTTP Interceptors

An HTTP Interceptor is a powerful feature in Angular that allows you to modify HTTP requests or responses globally, before they reach the server or the component that made the request.

Use Cases for Interceptors:

- **Adding Authorization Headers**: You can attach JWT tokens to the headers of all outgoing requests.
- **Logging**: Log all HTTP requests and responses for debugging or auditing.
- **Error Handling**: Globally catch errors from all HTTP requests.
- **Request Modification:** Modify requests before they are sent to the server (e.g., adding timestamps or modifying request body).

Example: Creating an HTTP Interceptor

```
import { Injectable } from '@angular/core';

import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable()

export class AuthInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    const token = localStorage.getItem('auth_token');

    const clonedRequest = req.clone({

      setHeaders: { Authorization: `Bearer ${token}` }

    });

    return next.handle(clonedRequest);

  }}
```

In this example:

- ❖ An AuthInterceptor is created that attaches a Bearer token to every outgoing HTTP request's headers.
- ❖ The intercept method clones the original request and adds an Authorization header with the token.

Registering the Interceptor

After creating the interceptor, you need to provide it in the AppModule:

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';

@NgModule({

  providers: [

    {

      provide: HTTP_INTERCEPTORS,

      useClass: AuthInterceptor,

      multi: true,

    }

  ]

})

export class AppModule {}
```

The multi: true option ensures that multiple interceptors can be used together.

INTRODUCTION

Java Platform, Enterprise Edition (Java EE) has revolutionized the way web applications are built. It evolved from the need for a robust, scalable, and secure platform for enterprise-level development. Java EE extends the capabilities of Java SE by providing APIs and runtime environments for distributed and web-based applications. It simplifies the development process by offering reusable components, integration with databases, and support for web services.

A diagram showing the core components like Servlets, JSP, EJB, and Web Services interacting with the database and client.

Java EE is the backbone of countless web applications across various domains such as e-commerce, healthcare, and finance. Its platform independence, coupled with features like security, scalability, and extensive community support, makes it a preferred choice for developers. Applications built on Java EE can cater to millions of users, ensuring high performance and reliability.

JAVA EE LAYERS

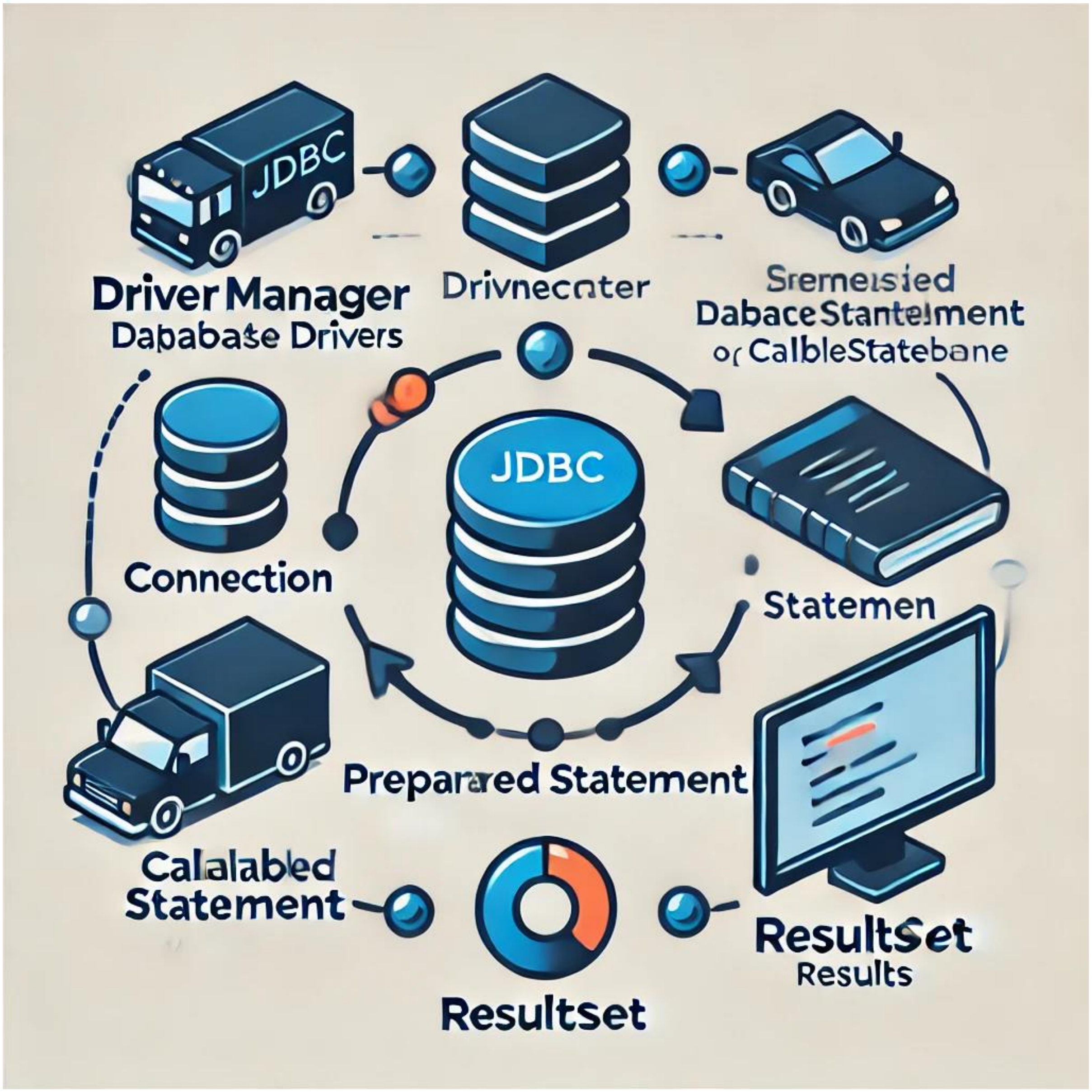
Depict layers such as **Presentation Layer** (JSP/Servlets), **Business Logic Layer** (EJB), and **Persistence Layer** (JDBC/ORM).

Servlets are the cornerstone of Java EE web development. They provide a simple yet powerful mechanism for building server-side components. Servlets enable developers to handle requests and generate responses dynamically. They form the basis of most web frameworks, offering flexibility and extensibility.

Servlet Request-Response Cycle

A flowchart showing the client sending a request, the servlet processing it, and returning a response to the browser.

JDBC (Java Database Connectivity) is an integral part of Java EE, enabling interaction between Java applications and relational databases. It allows developers to execute SQL queries, retrieve results, and manage transactions programmatically. JDBC ensures that Java EE applications can efficiently persist and fetch data from the database layer.



JDBC FLOW

Show how an application connects to a database through JDBC drivers, executes queries, and retrieves results.

JavaServer Pages (JSP) complements Servlets by allowing developers to embed Java code within HTML to create dynamic web pages. JSP is particularly effective for generating user interfaces while delegating business logic to Servlets or EJBs. Its integration with Expression Language (EL) and custom tags simplifies web development.

JSP IN MVC ARCHITECTURE

Illustrates JSP as the View layer, interacting with the Controller (Servlets) and the Model (Database via JDBC).

The true power of Java EE lies in the seamless integration of its components. By combining Servlets for request handling, JSP for presentation, and JDBC for database interaction, developers can build robust, maintainable, and scalable web applications. The framework’s modularity ensures adaptability to changing business needs.

Java EE Application Workflow

An end-to-end diagram showcasing a user request, servlet processing, database query execution via JDBC, and response generation using JSP.

This introduction establishes the foundation for understanding the Java EE ecosystem, its components, and its pivotal role in modern web development. Each diagram complements the textual content, providing a visual guide to the concepts discussed.

JDBC (JAVA DATABASE CONNECTIVITY)

Java Database Connectivity (JDBC) is a Java API that enables Java applications to interact with relational databases. JDBC provides methods for querying and updating data in a database, enabling developers to execute SQL statements from Java code. It abstracts the interaction between a Java application and a database, providing a standard interface for database operations.

JDBC allows for the following operations:

- **Connecting to a database**: Establishing a connection using a JDBC driver.

- **Executing SQL statements**: Using SQL queries to interact with a database, such as SELECT, INSERT, UPDATE, DELETE.
- **Retrieving and manipulating results**: Extracting data returned from the database using ResultSet, and manipulating data when necessary.
- **Handling transactions**: Managing transactions to ensure data consistency and integrity.

COMPONENTS OF JDBC

JDBC is composed of the following components:

1. **JDBC Drivers**: These are Java classes that implement the JDBC interfaces, enabling Java applications to connect to different types of databases.
2. **DriverManager**: A class that manages a list of database drivers. It helps in selecting an appropriate driver for database communication.
3. **Connection**: An interface representing an open connection to a database. It allows for creating statements, committing or rolling back transactions, and closing the connection.
4. **Statement**: An interface used to execute SQL queries against a database.
5. **PreparedStatement**: A subclass of Statement used for executing precompiled SQL queries with parameterized values, which are more efficient and secure than regular statements.
6. **CallableStatement**: Used to execute SQL stored procedures.
7. **ResultSet**: Represents the result of a query. It holds the data retrieved from the database, which can be accessed using various methods like next(), getInt(), and getString().

TYPES OF JDBC DRIVERS

There are four main types of JDBC drivers, each with its advantages and use cases:

1. **Type 1 – JDBC-ODBC Bridge Driver**: This driver uses the ODBC (Open Database Connectivity) driver to interact with the database. It is not commonly used anymore due to performance issues and lack of support in newer Java versions.
2. **Type 2 – Native-API Driver**: This driver converts JDBC calls into database-specific calls using native libraries. It is faster than Type 1 but still requires installation of database client software.
3. **Type 3 – Network Protocol Driver**: This driver communicates with a middleware server that translates JDBC calls into database-specific calls. It doesn't require a client installation on the machine where the application runs.
4. **Type 4 – Thin Driver**: This driver converts JDBC calls directly into database-specific calls using a pure Java implementation. It is the most efficient and widely used type of driver in modern applications.

ESTABLISHING A DATABASE CONNECTION

To interact with a database in JDBC, the first step is to establish a connection. This can be done using the DriverManager.getConnection() method, which accepts the database URL, username, and password. Here's an example:

Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "root", "password");

Once the connection is established, you can perform various database operations like querying or updating data.

JDBC API METHODS WITH EXAMPLES

1. **createStatement()**: Creates a Statement object used for executing SQL queries.

   ```
   Statement stmt = con.createStatement();
   ```

2. **executeQuery()**: Executes a SELECT SQL query and returns a ResultSet object.

   ```
   ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
   ```

3. **executeUpdate()**: Executes an INSERT, UPDATE, or DELETE SQL query.

   ```
   int rowsAffected = stmt.executeUpdate("INSERT INTO employees (name, position) VALUES ('John Doe', 'Manager')");
   ```

4. **setString()**: Used with PreparedStatement to set a string parameter in a query.

   ```
   PreparedStatement pstmt = con.prepareStatement("SELECT * FROM employees WHERE department = ?");
   pstmt.setString(1, "HR");
   ResultSet rs = pstmt.executeQuery();
   ```

5. **close()**: Used to close the ResultSet, Statement, and Connection objects to release resources.

   ```
   rs.close();
   stmt.close();
   con.close();
   ```

BEST PRACTICES FOR JDBC PROGRAMMING

1. **Use PreparedStatement over Statement**: Always use PreparedStatement to prevent SQL injection attacks and improve performance with parameterized queries.
2. **Always Close Resources**: Close the ResultSet, Statement, and Connection objects to avoid memory leaks and database connection issues.
3. **Handle Exceptions Properly**: Use proper exception handling using try-catch blocks to handle SQL exceptions. Always clean up resources in the finally block to ensure they are closed even if an exception occurs.
4. **Use Connection Pooling**: For better performance, use a connection pool to reuse database connections rather than opening a new one for every operation.
5. **Transactions Management**: Use transactions to ensure that multiple database operations are either fully completed or rolled back in case of an error.

SERVLETS

INTRODUCTION TO SERVLETS

A Servlet is a Java program that runs on a web server and extends the capabilities of a server. Servlets are primarily used to create dynamic web applications, handling client requests, processing them, and sending back the appropriate response.

Servlets are part of the Java EE (Enterprise Edition) and serve as the foundation for web applications in Java. A servlet is executed in response to client requests, typically through HTTP, and it processes the request, generates dynamic content, and returns the response to the client.

Example:

```
@WebServlet("/HelloServlet")
public class HelloServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        response.getWriter().println("Hello, World!");
    }
}
```

Servlets vs Other Server-side Technologies

While Java Servlets are powerful tools for creating dynamic web applications, there are other server-side technologies such as **PHP**, **ASP.NET**, and **Ruby on Rails**. Let's compare some key differences:

- **Performance**: Java Servlets are more scalable and perform better under heavy traffic compared to PHP and other scripting technologies.
- **Language**: Servlets are written in Java, a statically typed language, whereas PHP, Ruby, and ASP.NET use dynamically typed languages.
- **Integration**: Java Servlets integrate seamlessly with other Java-based technologies such as JDBC for database connectivity, while PHP integrates well with MySQL.
- **Concurrency**: Servlets are better equipped for handling multiple requests concurrently due to their multi-threaded nature.

LIFECYCLE OF A SERVLET

The lifecycle of a servlet is controlled by the **Servlet Container** (also known as the web container). It defines how the servlet is loaded, initialized, serviced, and destroyed. There are four main stages in the lifecycle:

1. **Loading and Instantiation**: The servlet is loaded when the first request is made.

2. **Initialization**: The init() method is called once after the servlet is instantiated.
3. **Request Handling**: The service() method processes client requests. This method is called multiple times, once for each request.
4. **Destruction**: The destroy() method is invoked when the servlet is removed from service, usually when the server shuts down.

Example:

```
public class LifeCycleServlet extends HttpServlet {
   public void init() {
      System.out.println("Servlet Initialized");
   }
   public void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
      response.getWriter().println("Request Received");
   }
   public void destroy() {
      System.out.println("Servlet Destroyed");
   }
}
```

CORE SERVLET INTERFACES AND CLASSES

Servlets use several key interfaces and classes. Here are the core components:

1. **HttpServlet**: The base class for handling HTTP requests. It simplifies handling HTTP-specific operations.
2. **HttpServletRequest**: Represents the request from the client, containing all the information like parameters, headers, and cookies.

3. **HttpServletResponse**: Represents the response sent to the client, including data such as the status code, headers, and content.
4. **ServletConfig**: Contains initialization parameters for the servlet.
5. **ServletContext**: Provides servlet-level context for interaction with the container.

HANDLING REQUESTS AND RESPONSES

Servlets handle requests and responses through the HttpServletRequest and HttpServletResponse objects. The doGet() and doPost() methods are used to handle HTTP GET and POST requests, respectively.

Example of Handling a Request:

```
@WebServlet("/RequestHandler")
public class RequestHandlerServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
       String name = request.getParameter("name");
       response.getWriter().println("Hello, " + name);
   }
}
```

SERVLET CONFIGURATIONS (ANNOTATIONS AND WEB.XML)

Servlets can be configured using annotations or in the web.xml file.

1. **Using Annotations**:

```
@WebServlet("/HelloWorld")
```

```
public class HelloWorldServlet extends HttpServlet {
    // Servlet code here
}
```

2. **Using web.xml**: In the web.xml file, the servlet is defined as follows:

```
xml
<servlet>
    <servlet-name>HelloWorldServlet</servlet-name>
    <servlet-class>com.example.HelloWorldServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>HelloWorldServlet</servlet-name>
    <url-pattern>/HelloWorld</url-pattern>
</servlet-mapping>
```

Advanced Topics: Filters, Listeners

1. **Filters**: Filters are used to modify request and response objects, such as logging, authentication, or modifying content before sending it to the client.
2. **Listeners**: Listeners are objects that listen to events in a web application (such as session creation or destruction) and respond accordingly.

<u>JSP (JAVASERVER PAGES)</u>JSP is a technology used to develop dynamic web pages. JSP allows embedding Java code directly into HTML, making it easy to

create dynamic content. JSP files have the extension .   and are compiled into Servlets by the web container.

<u>Example:</u>

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<html>
<head>
   <title>Hello World</title>
</head>
<body>
   <h1>Hello World from    !</h1>
</body>
</html>
```

<u>JSP vs Servlets</u>

- **JSP** is primarily used for presenting data (view layer) and generating dynamic content using HTML and Java.
- **Servlets** are used for handling business logic (controller layer) and managing HTTP requests/responses.

JSP offers a simpler syntax, whereas Servlets require more lines of code to generate the same output.

JSP Elements: Directives, Scripting Elements, Actions

1. **Directives**: Provide configuration information to the JSP container. They appear at the top of the page.

```
<%@ page language="java" contentType="text/html" %>
```

2. **Scripting Elements**: Allow embedding Java code in JSP.
    - **Declarations**: Declare variables and methods.

      ```
      <%! int count = 0; %>
      ```

    - **Scriptlets**: Embed Java code within HTML.

      ```
      <%
        int count = 1;
        out.println("Count: " + count);
      %>
      ```

    - **Expressions**: Output Java expressions directly.

      ```
      <%= "Hello, World!" %>
      ```

3. **Actions**: Predefined tags for controlling the behavior of the page.
    - **jsp:include**: Includes content from another page.
    - **jsp:forward**: Forwards the request to another page.

EXPRESSION LANGUAGE (EL)

EL simplifies the interaction between JSP and Java objects by providing an easy way to access JavaBeans properties and expressions.

Example:

```
${user.name}
```

EL automatically resolves the value of user.name from a JavaBean.

JSP AND MVC ARCHITECTURE

JSP is often used as the **View** in the **Model-View-Controller (MVC)** architecture. Servlets handle the **Controller** logic, processing the request and forwarding it to the appropriate JSP page, which renders the **View**.

JSP with Custom Tags

Custom tags in JSP allow you to define reusable components and extend JSP's functionality. You can create your own tags for specific actions, which makes the code cleaner and more modular.

1. Frontend (UI Layer)

The frontend represents what the user sees and interacts with. In your case, it's the **HTML/CSS page** that handles the structure and appearance of the Admin Page.

EXAMPLE: ADMIN PAGE

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Admin Page</title>
    <style>
        /* Styling for header, navigation, etc. */
        /* Flexbox Layout and responsiveness */
        /* ... */
    </style>
</head>
<body>
    <header class="header">
        <h1 class="logo">Logo</h1>
        <div class="admin-header">
            <span>Admin</span>
        </div>
        <ul class="nav">
            <li><a href="Adminuploadfile.jsp">Upload Data</a></li>
            <li><a href="#">Employee Status</a></li>
            <li><a href="#">Industry Data</a></li>
            <li><a href="#">Testing Analysis</a></li>
            <li><a href="#">Pasting</a></li>
            <li><a href="#">Download Data</a></li>
            <li><a href="#">Logout</a></li>
        </ul>
    </header>
</body>
</html>
```

- **Responsiveness**: You used Flexbox layout to make the page responsive. Good for different screen sizes.
- **Navigation Links**: You have links such as "Upload Data", "Employee Status", etc., which are important for the admin interface.
- **Styling**: The page has custom styles for the header and navigation to make it visually appealing.

2. Business Layer (Servlet Layer)

The business layer contains logic that processes the requests and handles communication between the frontend and the backend (database).

<u>Example: Login Servlet (login.java)</u>

```
package Adminpage;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/login")
public class login extends HttpServlet {
   private static final long serialVersionUID = 1L;
   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
       String username = request.getParameter("username");
       String password = request.getParameter("password");
       // Simple logic to authenticate and redirect to the admin homepage.
       // In a real-world application, you would check the credentials from the database.
       response.sendRedirect("adminhomepage.html");
   }
}
```

- **Login Authentication**: In this example, the login servlet captures the username and password from the request and performs some authentication logic.
    - **Improvement**: Implement real authentication using a database to check user credentials instead of just redirecting.

3. DATABASE LAYER

The database layer handles the connection to the database, providing methods to retrieve or insert data.

Example: Database Connection (Dbconn.java)

java
Copy code

```
package dbconnection;

import java.sql.Connection;
import java.sql.DriverManager;

public class Dbconn {

    static Connection con;

    public static Connection getconnection() {
        try {
            // Load the database driver
            Class.forName("com.mysql.jdbc.Driver");

            // Establish a connection to the database
            con = DriverManager.getConnection("jdbc:mysql://localhost:3306/saloon", "root",
"root");
        } catch (Exception e) {
            e.printStackTrace();
        }
    return con;
 }}
```

- **Database Connection**: The Dbconn class is responsible for establishing a connection to the MySQL database using JDBC.
    - **Improvement**: You can manage connections using a connection pool (like Apache DBCP or HikariCP) for better performance and resource management in a production environment.

SUGGESTED FOLDER STRUCTURE (MVC PATTERN)

To improve organization and scalability, you can structure the application using the **Model-View-Controller (MVC)** pattern:

- **Model (Database Layer)**: Represents data and the logic for interacting with the database.
    - Dbconn.java
    - User.java (For user-related actions in the database)
- **View (Frontend)**: Contains all the HTML/JSP pages and frontend logic.
    - adminPage.jsp
    - login.jsp
    - adminhomepage.jsp
- **Controller (Servlet Layer)**: Handles user requests and communicates with the Model.
    - LoginServlet.java
    - AdminDataServlet.java (for handling admin data requests)

Example for Handling the Business Layer (Login Validation)

You can improve the login servlet by validating the credentials from the database. Here's an updated example:

<u>LOGINSERVLET.JAVA</u>

```
package Adminpage;
import dbconnection.Dbconn;
import java.io.IOException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;
   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
      String username = request.getParameter("username");
      String password = request.getParameter("password");
      Connection conn = Dbconn.getconnection();
      try {
        // Prepare SQL query to validate login credentials
        String query = "SELECT * FROM users WHERE username = ? AND password = ?";
        PreparedStatement pst = conn.prepareStatement(query);
        pst.setString(1, username);
        pst.setString(2, password);
        ResultSet rs = pst.executeQuery();
        if (rs.next()) {
            // Successful login, redirect to admin homepage
            response.sendRedirect("adminhomepage.jsp");
        } else {
            // Invalid login, redirect to login page with error
            response.sendRedirect("login.jsp?error=invalid");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}}
```

- **Database Query**: This updated servlet checks if the username and password match any record in the users table.
- **Error Handling**: Redirects back to the login page if authentication fails with an error message.

SUMMARY OF LAYERS AND EXAMPLE METHODS:

1. **Frontend (View Layer)**:
    - HTML/CSS for user interface.
    - Responsive design with Flexbox.
    - Links for navigation like Upload Data, Logout, etc.

2. **Business Layer (Controller Layer)**:
    - Servlets to process user requests, such as login and redirect to different pages.
    - Logic to handle user authentication and data submission.
3. **Database Layer (Model Layer)**:
    - Java class to manage database connection (JDBC).
    - SQL queries to interact with the database (CRUD operations).

JAVA WEB APPLICATION

- A **web application** is software that runs on web browsers and is accessible over the internet.

- It provides a **graphical user interface (GUI)** for users to interact with and performs various business and data management tasks.

WHAT IS SPRING FRAMEWORK:

- Spring is a lightweight, open-source Java framework.
- Created by Rod Johnson in 2003, designed for building enterprise-level applications.
- Known as a "framework of frameworks,"
- It integrates with technologies like Hibernate, Struts, and JSF.
- Its core features include modules for IOC (Inversion of Control), AOP (Aspect-Oriented Programming), DAO, Context, and Web MVC.

WHY USE SPRING:

- Spring simplifies Java application development by promoting POJO-based programming and good practices.

- It supports building lightweight, modular, and testable applications while integrating seamlessly with existing frameworks like ORM tools, logging frameworks, and web technologies.

Key Features:

- **POJO-Based Development**: No need for complex EJB containers; works with lightweight servers like Tomcat.
- **Modular Design**: Use only the required components.
- **Integration-Friendly**: Works with popular tools like Hibernate, Quartz, and JEE.
- **Testability**: Simplifies testing with dependency injection.
- **Web MVC Framework**: A robust alternative to frameworks like Struts.
- **Centralized Exception Handling**: Converts technology-specific exceptions into unchecked exceptions.
- **Lightweight**: Ideal for systems with limited resources.

KEY CHARACTERISTICS OF MODERN WEB APPLICATIONS

1. **Cross-platform Accessibility:** Accessible across devices and platforms via web browsers.
2. **Scalability:** Ability to handle increasing user loads effectively.
3. **Integration:** Interoperability with third-party services via APIs.
4. **Security:** Implements robust authentication, authorization, and data protection mechanisms.
5. **User-Centric Design:** Offers responsive and intuitive user interfaces.

HISTORY OF WEB DEVELOPMENT IN JAVA

1. Early Beginnings (1995-1997)

- **Java 1.0** was launched in 1995, laying the foundation for Java-based web development.
- The **Servlet API 1.0** was introduced in 1997, enabling dynamic web applications.

- Early Java web applications were primarily built using **Servlets** and **Java Server Pages (JSP)**.

Servlets:

- Servlets are Java classes that run on a web server, handling HTTP requests and generating responses.
- They extend web server capabilities, enabling tasks such as:
  - ➢ Form submission processing
  - ➢ User authentication
  - ➢ Session management
  - ➢ Database interactions

JSP (Java Server Pages):

- JSP allows the seamless combination of HTML and Java code in a single file.
- At runtime, JSP files are compiled into servlets, making them an efficient tool for generating dynamic content.
- Key uses of JSP include:
  - ➢ Displaying dynamic data
  - ➢ Integrating JavaBeans for data storage
  - ➢ Including other JSP or HTML files
  - ➢ Using **JSTL (Java Server Pages Standard Tag Library)** for common tasks

2. Java 2 Enterprise Edition (J2EE) Era (1999-2003)

- **J2EE 1.2**, released in 1999, marked a major milestone in enterprise-level Java development.

- **Enterprise JavaBeans (EJB)** were introduced, facilitating the development of complex, distributed architectures.

3. Struts and Spring Frameworks (2000-2005)

- The **Apache Struts 1.0** framework debuted in 2000, providing a structured approach to web application development.
- The **Spring Framework 1.0**, launched in 2004, introduced simpler methods for creating robust web applications.

4. Ajax and Web 2.0 (2005-2008)

- The rise of **Asynchronous JavaScript and XML (Ajax)** enabled dynamic, interactive web applications.
- Rich Internet Applications (RIAs) became widely popular.
- **Java Server Faces (JSF)**, a component-based web framework, was introduced to streamline UI development.

5. Java EE 5 and 6 (2006-2010)Java EE 5, released in 2006, introduced annotations and dependency injection, significantly simplifying development.

- **Java EE 6** continued this trend, offering enhanced tools for creating scalable applications.

6. Micro services and DevOps Era (2015-Present)

- The adoption of **micro services architecture** revolutionized application design, promoting modular and scalable systems.
- **DevOps practices** became integral to software development and deployment.

- **Java EE 8**, released in 2017, introduced modern enhancements, including support for cloud-native applications.
- In 2018, **Jakarta EE** succeeded Java EE, continuing its legacy under a new name.

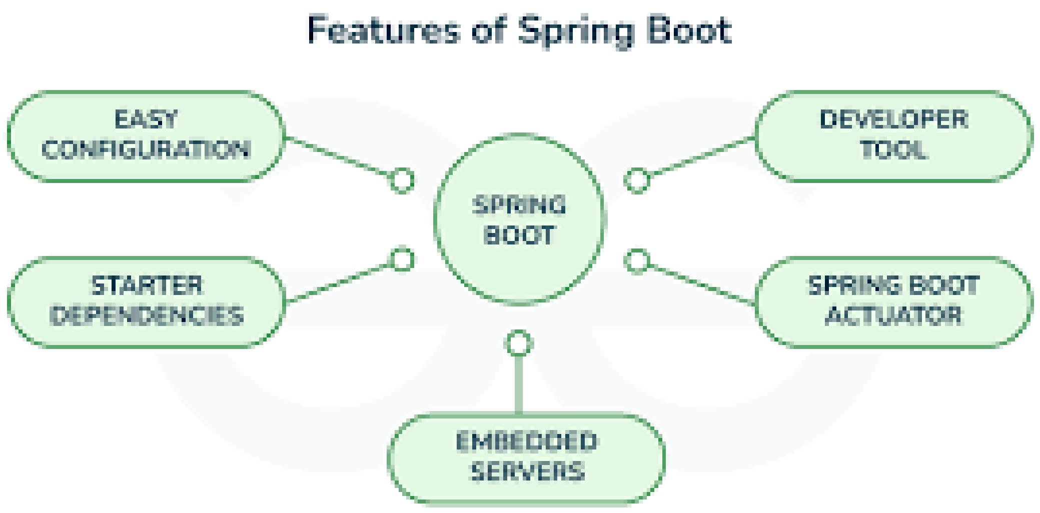
This evolution highlights Java's versatility and enduring relevance in web development.

SPRING BOOT

- ❖ Spring Boot is an open-source Java framework designed to simplify the development and deployment of Java applications.
- ❖ Built by the Pivotal Team, it enables Rapid Application Development (RAD) with minimal configuration.
- ❖ It combines the Spring Framework with embedded servers, eliminating the need for extensive XML configuration.
- ❖ This tutorial is ideal for beginners and professionals, covering basic to advanced topics such as REST APIs, Micro services, Kafka, and Data JPA integration.

KEY FEATURES OF SPRING BOOT

- ❖ Simplifies setup and configuration for web and enterprise applications
- ❖ Supports Micro services architecture.
- ❖ Reduces development and testing time.
- ❖ Eliminates XML configuration with a convention-over-configuration approach.



LAYERS OF A WEB APPLICATION

The modern layered architecture consists of three primary layers:

PRESENTATION LAYER

- **Role:** Responsible for the user interface (UI) and user interaction.
- **Technologies:**
  - **Languages:** HTML5, CSS3, JavaScript (ES6+).
  - **Frameworks/Libraries:** Angular, React, Vue.js, Bootstrap, Tailwind CSS.
- **Modern Additions:**
  - **Web Components:** Reusable custom HTML elements encapsulating functionality.

- **Progressive Web Apps (PWAs):** Provide offline capabilities, faster load times, and native app-like features.

Example:

- A modern Spring Boot application often uses **React** for the front-end.
- A simple React component can look like this:

```
function WelcomeComponent({ username }) {

  return <h1>Welcome, {username}!</h1>;}
```

BUSINESS LOGIC LAYER

- **Role:** Manages application logic, processes requests, and connects the front-end with the database.
- **Technologies:**
  - **Languages:** Java (with Spring Framework or Spring Boot), Python (Flask/Django), Kotlin.
  - **Approaches:**
    - **RESTful APIs:** For scalable, stateless interactions.
    - **GraphQL:** For flexible and efficient data queries.
    - **gRPC:** For high-performance, cross-language communication.

Example in Spring Boot:

```
@RestController

@RequestMapping("/api")

public class UserController {
```

```
private final UserService userService;



public UserController(UserService userService) {

    this.userService = userService;

}



@GetMapping("/users/{id}")

public ResponseEntity<User> getUserById(@PathVariable Long id) {

    return ResponseEntity.ok(userService.findById(id));

}  }
```

3. Data Storage Layer

- **Role:** Handles data persistence and retrieval using relational or non-relational databases.
- **Technologies:**
    - **Relational Databases:** MySQL, PostgreSQL, Oracle DB.
    - **NoSQL Databases:** MongoDB, Redis, DynamoDB.
    - **Modern Additions:**
        - **Data Lakes:** Store structured and unstructured data for analytics.
        - **Graph Databases:** Neo4j for relationship-focused data models.
        - **Cloud Databases:** Managed services like AWS RDS, Google Firestore.

Example in Spring Boot:

```
@Repository
```

```
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByEmail(String email);

}
```

NEW DESIGN PATTERNS AND PRINCIPLES:

BEAN :

- In Spring Boot, a bean is an object that is managed by the Spring IoC (Inversion of Control) container.
- Beans are the backbone of a Spring-based application, and they represent a reusable component or a service that can be used throughout the application.

INVERSION OF CONTROL (IOC):

- **Concept:** Delegates object creation and lifecycle management to the Spring IoC container.
- **Enhancement:** Combined with **Dependency Injection** for modular, testable designs.

Modern Example: Using Constructor Injection:

```
@Component

public class NotificationService {

    public String notifyUser(String message) {

        return "Notification: " + message;

    }
}

@Service

public class UserService {

    private final NotificationService notificationService;

    @Autowired

    public UserService(NotificationService notificationService) {

        this.notificationService = notificationService;

    }

    public String welcomeUser(String username) {

        return notificationService.notifyUser("Welcome " + username);

    }

}
```

DEPENDENCY INJECTION (DI) :

Spring Boot provides powerful dependency injection features using its Spring Framework.

It supports all three types of DI:

- ❖ Constructor Injection,
- ❖ Setter Injection
- ❖ Field Injection.

1. **Constructor Injection:**

This is the recommended approach for DI in Spring, especially for mandatory dependencies.

Example:

Service Class (Dependency):

```
package com.example.demo.service;


import org.springframework.stereotype.Service;


@Service
public class EngineService {
    public void startEngine() {
        System.out.println("Engine started!");
    }
}
```

@Service Annotation

The @Service annotation indicates that the class is a service component in the Spring application.

Role of @Service:

- Marks the class for automatic discovery by Spring's component scanning.
- Ensures the class is managed as a Spring bean, making it eligible for dependency injection in other components like controllers or other services.

Dependent Class:

```
package com.example.demo.car;

import com.example.demo.service.EngineService;
import org.springframework.stereotype.Component;

@Component
public class Car {
    private final EngineService engineService;

    // Constructor Injection
    public Car(EngineService engineService) {
        this.engineService = engineService;
    }

    public void drive() {
        engineService.startEngine();
        System.out.println("Car is driving...");
    }
}
```

@Component Annotation

- Role: Indicates that the Car class is a Spring-managed component.
- During component scanning, Spring identifies this class and registers it as a bean in the Application Context.
- This makes the Car object available for dependency injection wherever needed.

Main Application:

```
package com.example.demo;

import com.example.demo.car.Car;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class DemoApplication implements CommandLineRunner {

        private final Car car;

  @Autowired

  public DemoApplication(Car car) {

    this.car = car;

  }
```

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

        SpringApplication.run(DemoApplication.class, args);

    }

@Override

    public void run(String... args) throws Exception {

  car.drive();

}

}
```

```
Output:            Engine started!

                   Car is driving..
```

Constructor Injection:

❖ Spring automatically identifies the dependency (EngineService) and injects an instance of it into the Car object when it creates the Car bean.

❖ The engineService field is assigned the value provided by Spring.

Why Constructor Injection?

❖ **Mandatory Dependencies**: Ensures that the Car class cannot be instantiated without providing the required EngineService dependency.

❖ **Immutability**: Dependencies are final, which prevents modification after object creation

❖ **Testability:** Makes it easier to provide mock or alternative implementations during testing

2.SETTER INJECTION

- ❖ This approach is used for optional dependencies. one of the ways to inject dependencies into a class.
- ❖ In this approach, Spring uses the setter methods of a class to inject dependencies after the object is created.
- ❖ Setter injection allows for more flexibility as it allows the injection of dependencies after the object’s construction, unlike constructor injection which is required during object creation.

Example:

Service Class (Dependency):

```
package com.example.demo.service;

import org.springframework.stereotype.Service;

@Service

public class EngineService {

    public void startEngine() {

        System.out.println("Engine started!");      } }
```

Dependent Class:

```
package com.example.demo.car;

import com.example.demo.service.EngineService;

import org.springframework.stereotype.Component;

@Component

public class Car {
    private EngineService engineService;

    // Setter Injection

    @Autowired

    public void setEngineService(EngineService engineService) {

        this.engineService = engineService;

}

    public void drive() {

        if (engineService != null) {

            engineService.startEngine();

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

        } else {

            System.out.println("No engine available to drive.");

        }

    }

}
```

Main Application:

```
package com.example.demo;

import com.example.demo.car.Car;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.SpringApplication;

@SpringBootApplication

public class DemoApplication implements CommandLineRunner {

    @Autowired

    private Car car;

    public static void main(String[] args) {

        SpringApplication.run(DemoApplication.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

        car.drive();

    }

}
```

Output:

```
Engine started!

Car is driving...
```

3. FIELD INJECTION:

- ❖ Field injection directly injects dependencies into fields.
- ❖ However, it is not recommended as it’s harder to test and violates the Single Responsibility Principle.

Example:

Service Class (Dependency):

```
package com.example.demo.service;

import org.springframework.stereotype.Service;

@Service

public class EngineService {

    public void startEngine() {

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

    }

}
```

Dependent Class:

```
package com.example.demo.car;

import com.example.demo.service.EngineService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Component

public class Car {

    @Autowired

    private EngineService engineService; // Field Injection

    public void drive() {
        engineService.startEngine();

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

    } }
```

Main Application:

```
package com.example.demo;


import com.example.demo.car.Car;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.SpringApplication;


@SpringBootApplication

public class DemoApplication implements CommandLineRunner {


    @Autowired

    private Car car;
    public static void main(String[] args) {

        SpringApplication.run(DemoApplication.class, args);

    }



    @Override

    public void run(String... args) throws Exception {

        car.drive();

    }

}
```

Output:

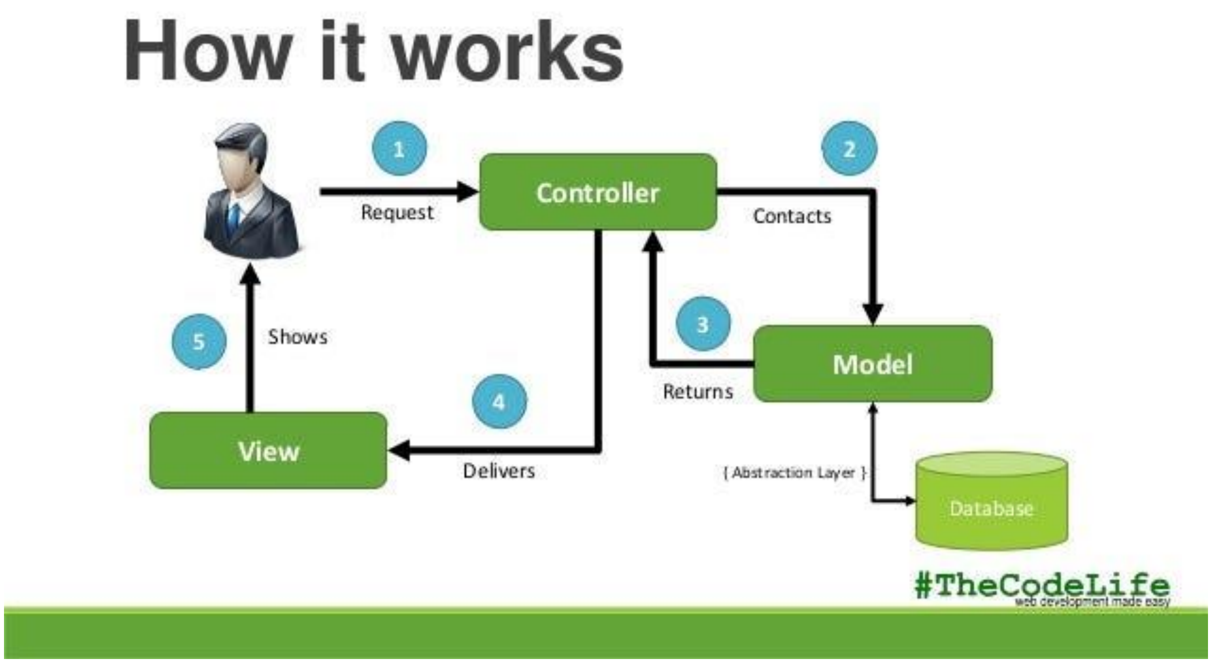
```
Engine started!

Car is driving...
```

Summary of Usage

Type	When to Use
Constructor Injection	For mandatory dependencies (Recommended).
Setter Injection	For optional dependencies or when flexibility is needed.
Field Injection	Should be avoided; use sparingly for legacy systems.

MVC WITH SPRING BOOT

- **Model-View-Controller** pattern is implemented seamlessly using Spring Boot.
  - **Model:** Managed by **JPA Entities**.
  - **View:** Can use **Thymeleaf**, **Mustache**, or **React/Angular** front-end.
  - **Controller:** Manages the request/response lifecycle.



POJO CLASES :

- In Spring Boot, POJO classes are commonly used as data carriers or domain objects, which represent and hold data throughout different layers of the application.

- These classes work seamlessly with Spring Boot due to its design principles like dependency injection, loose coupling, and data-binding.

ROLES OF POJO IN SPRING BOOT:

- **Entity Representation:**

POJOs are used to represent entities in database operations, typically annotated with JPA/Hibernate annotations like @Entity.

- **Data Transfer Objects (DTOs):**

POJOs can serve as DTOs to transfer data between layers like controllers, services, and repositories.

- **Configuration and Properties Mapping:**

POJOs can map application properties using @ConfigurationProperties or @Value.

- **Request and Response Objects:**

POJOs represent the body of HTTP requests and responses in REST APIs.

- **Dependency Injection:**

Spring Boot uses POJOs as services or components, injecting them where needed.

ANNOTATIONS IN SPRING BOOT :

- Spring Boot leverages a wide range of annotations to provide declarative programming features, streamline configurations, and enhance productivity.

- Below is a list of commonly used Spring Boot annotations, their purposes, and additional annotations for advanced use cases.

- ### CORE ANNOTATIONS :

@SpringBootApplication:

Combines ***@Configuration, @EnableAutoConfiguration, and @ComponentScan*** to simplify Spring Boot setup.

@Component: Marks a class as a generic Spring-managed component.

@Service: Specialization of @Component for service layer beans.

@Repository: Specialization of @Component for persistence layer beans.

Automatically translates database-related exceptions

@Controller:	Specialization of @Component to define a Spring MVC controller.
@RestController:	Combines ***@Controller and @ResponseBody*** to return JSON/XML
@Bean:	Declares a method that produces a bean to be managed by the Spring container.

DEPENDENCY INJECTION ANNOTATIONS :

@Autowired:	Injects dependencies automatically.
@Qualifier:	Specifies which bean to inject when multiple beans of the same type exist.
@Primary	Marks a bean as the primary candidate for injection when multiple beans of the same type exist.
@Lazy:	Marks a bean to be initialized lazily (on first access).

@Value:	Injects values from application.properties or application.yml.
@Scope:	Defines the scope of a bean (singleton, prototype, etc.).

Web Annotations :

@RequestMapping:	Maps HTTP requests to handler methods or classes.
@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping:	Specialized annotations for specific HTTP methods.
@PathVariable:	Extracts values from URI templates.
@RequestParam:	Extracts query parameters.
@RequestBody:	Maps the HTTP request body to a method parameter.

@ResponseBody: Indicates that the return value of a method is serialized and sent as the response body.

@CrossOrigin: Enables Cross-Origin Resource Sharing (CORS) for a method or class.

VALIDATION AND DATA BINDING :

@Valid: Triggers validation on a request or response object.

@NotNull, @NotBlank, @Size, @Pattern, etc.:

Bean validation annotations to enforce constraints on fields.

@RequestPart: Maps a multipart request (e.g., file upload) to a method parameter.

@ModelAttribute: Binds a model attribute to a method parameter or return value. Persistence and Transaction Management

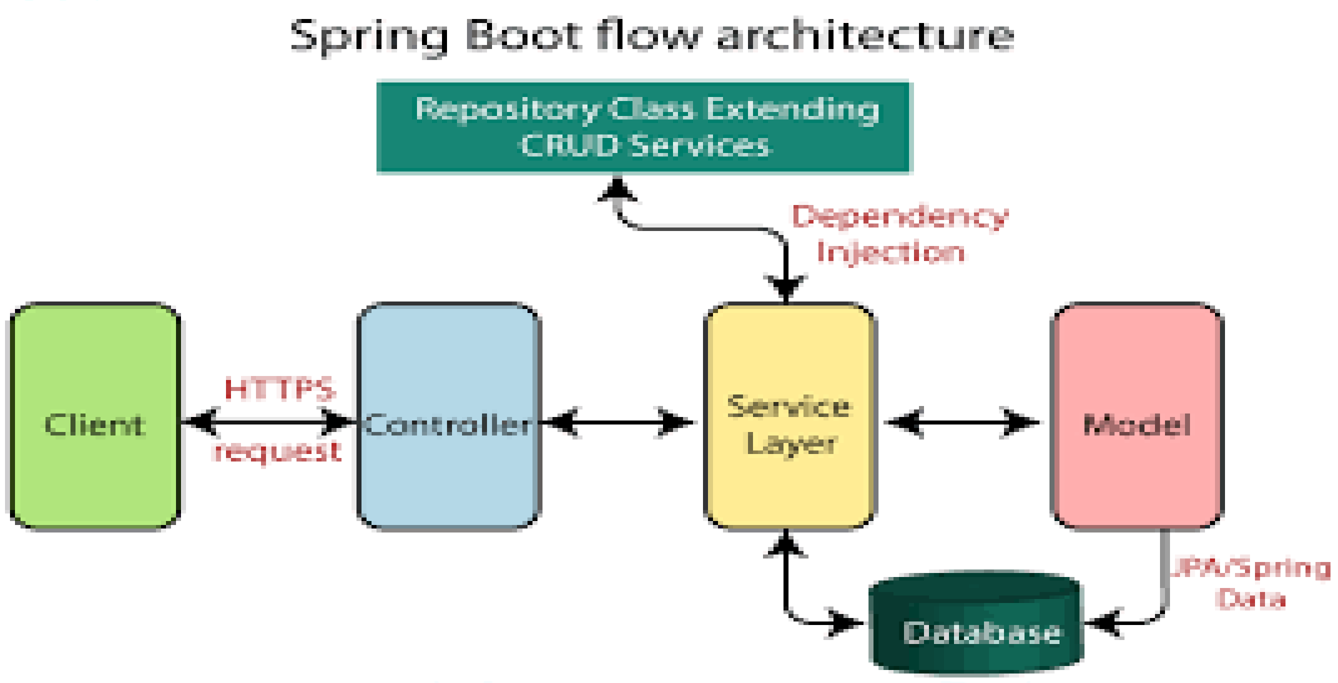
@Entity: Marks a class as a JPA entity (mapped to a database table).

@Id: Marks a field as the primary key.

@GeneratedValue	Configures how the primary key value is generated (e.g., AUTO, IDENTITY).
@Table:	Specifies the table name and other table-level attributes.
@Column:	Configures column-level details.
@Transactional:	Manages transaction boundaries on methods or classes.
@ManyToOne, @OneToMany, @OneToOne, @ManyToMany:	Define relationships between entities.

STEPS TO CREATE A BASIC CRUD APPLICATION:

1. Create a POJO Class
2. Create Repository
3. Create Interface and service class for the implementations
4. Create Controller Class
5. Run the Application



How POJO Works in Spring Boot:

1. Mapping Entities to Databases

When used with JPA, POJO classes are annotated as entities, enabling ORM (Object Relational Mapping)

Example:

```
import jakarta.persistence.Entity;

import jakarta.persistence.Id;

@Entity

public class User {

    @Id

    private Long id;
    private String name;

    private int age;

    public User() {}

    public User(Long id, String name, int age) {

        this.id = id;

        this.name = name;

        this.age = age; }  // Getters and Setters

}
```

<u>HOW IT WORKS:</u>

- ❖ The @Entity annotation maps this POJO to a database table.
- ❖ Fields are mapped to columns, and Spring Boot uses this POJO for CRUD operations through Jpa Repository.

<u>HOW JPA WORKS IN SPRING BOOT</u>

- ❖ Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation.
- ❖ JPA is a specification which specifies how to access, manage and persist information/data between java objects and relational databases
- ❖ It provides a standard approach for ORM, Object Relational Mapping. Spring Boot provides a seemless integration with JPA.
- ❖ JpaRepository provides many built-in methods for database operations:

CRUD OPERATIONS:

Method	Description
save(S entity)	Saves or updates an entity in the database.
findById(ID id)	Finds an entity by its primary key.
findAll()	Retrieves all entities.
deleteById(ID id)	Deletes an entity by its primary key.
delete(S entity)	Deletes a specific entity.

PAGINATION AND SORTING:

Method	Description
findAll(Sort sort)	Retrieves all entities sorted by a specified criterion.
findAll(Pageable pageable)	Retrieves a paginated list of entities.

QUERY EXECUTION:

Method	Description
existsById(ID id)	Checks if an entity with the given ID exists.
count()	Counts the total number of entities.

JPQL QUERIES:

Use the @Query annotation for complex queries.

Example:

```
@Query("SELECT u FROM User u WHERE u.email = :email")

User findUserByEmail(@Param("email") String email);
```

NATIVE SQL QUERIES:

To execute raw SQL:

```
@Query(value = "SELECT * FROM users WHERE email = :email", nativeQuery = true)

User findUserByEmailNative(@Param("email") String email);
```

REST API :

- ❖ A REST API (Representational State Transfer Application Programming Interface) allows communication between client and server applications over HTTP.
- ❖ It's widely used to create web services for exchanging data, usually in JSON or XML format.

KEY FEATURES OF REST API:

- ❖ **Stateless:** Each request contains all information needed; the server does not store session data.

- **Resource-Oriented:** Everything (like a user, product, etc.) is treated as a resource with unique URIs.

Standard HTTP Methods:

- GET: Retrieve data.
- POST: Create new data.
- PUT: Update existing data.
- DELETE: Remove data.

HTTP Status Codes: Provides responses like 200 OK, 404 Not Found, and 201 Created.

SPRING BOOT REST CONTROLLER :

In Spring Boot, REST APIs are built using controllers with annotations like:

@RestController:

Indicates it's a RESTful service.

@RequestMapping:

Maps a base URI to the controller or its methods.

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping:

Map specific HTTP methods to Java methods.

HERE ARE SOME COMMONLY USED SPRING BOOT ANNOTATIONS IN REST APIS :

1. @PathVariable

- Purpose: Extracts values from the URI path.

- Use Case: For dynamic parts of the URL.

Example:

```
@GetMapping("/users/{id}")

public User getUser(@PathVariable Long id) {

    // Extracts the 'id' from the URL: /users/1

    return userService.getUserById(id);

}
```

2. **@RequestBody**

- Purpose: Maps the HTTP request body to a Java object.

- Use Case: For sending JSON or XML data in POST or PUT requests.

Example:

```
@PostMapping("/users")

public User createUser(@RequestBody User user) {

    // Converts incoming JSON payload into a User object

    return userService.createUser(user);

}
```

3. **@RequestParam**

- Purpose: Extracts query parameters from the URL.

- Use Case: For optional parameters in requests.

Example:

```
@GetMapping("/users")

public List<User> getUsers(@RequestParam String role) {

    // Extracts 'role' from the query string: /users?role=admin

    return userService.getUsersByRole(role);

}
```

4. @RequestHeader

- Purpose: Extracts values from HTTP headers.
- Use Case: For custom or required headers in requests.

Example:

```
@GetMapping("/secure-data")

public String getSecureData(@RequestHeader("Authorization") String token) {

    // Reads 'Authorization' header from the request

    return securityService.validateToken(token);

}
```

5. @ResponseBody

- Purpose: Converts Java objects into HTTP response body (usually JSON or XML).
- Use Case: Automatically added when using @RestController.

Example:

```
@GetMapping("/users/{id}")
```

```
@ResponseBody

public User getUser(@PathVariable Long id) {

    // Automatically serializes User object to JSON

    return userService.getUserById(id);

}
```

6. @RequestMapping

- ❖ Purpose: Maps a URL to a controller or its methods. Can handle multiple HTTP methods.
- ❖ Use Case: For basic mappings or when using method-specific annotations like @GetMapping.

Example:

```
@RequestMapping(value = "/users", method = RequestMethod.GET)

public List<User> getAllUsers() {

    return userService.getAllUsers();

}
```

7. @ResponseStatus

- ❖ Purpose: Specifies the HTTP status for the response.
- ❖ Use Case: To return custom HTTP status codes.

Example:

```
@DeleteMapping("/users/{id}")

@ResponseStatus(HttpStatus.NO_CONTENT)

public void deleteUser(@PathVariable Long id) {
    userService.deleteUserById(id);
```

8. @CrossOrigin

- Purpose: Enables Cross-Origin Resource Sharing (CORS).
- Use Case: For allowing API calls from different domains.

Example:

```
@CrossOrigin(origins = "http://example.com")

@GetMapping("/users")

public List<User> getUsers() {

    return userService.getAllUsers();

}
```

9. @Valid

- Purpose: Validates request bodies based on constraints defined in the entity class.
- Use Case: For automatic validation of inputs.

Example:

```
@PostMapping("/users")

public User createUser(@Valid @RequestBody User user) {

    // Validates user fields (e.g., not null, email format, etc.)

    return userService.createUser(user);

}
```

10. @ExceptionHandler

- Purpose: Handles specific exceptions globally or for a specific controller.
- Use Case: To return custom error responses.

Example:

```
@ExceptionHandler(UserNotFoundException.class)

@ResponseStatus(HttpStatus.NOT_FOUND)

public String handleUserNotFound(UserNotFoundException ex) {

    return ex.getMessage();

}
```

These annotations make building REST APIs in Spring Boot more efficient by managing request and response data effortlessly!

VCODEZ
INNOVATING IDEAS

MASTERING JAVA PROGRMMING

Learn Java with this easy-to-follow e-book featuring real-world projects and simple, clear examples. Perfect for building your coding skills step by step!

SYNOPSIS:

This comprehensive e-book will take you from basic ideas to advanced knowledge in Java programming! Whether you're an a beginner interested in coding or a developer looking to improve your abilities, this resource blends straightforward explanations with practical examples to provide a smooth learning experience. But that is not all! We go beyond Java and provide critical insights into HTML and CSS to establish a solid basis for full-stack web development. This e-book, packed with professional advice, best practices, and interactive step-by-step guidance, is the perfect companion unlocking the power of programming.

VCodez Innovating Ideas

At VCodez, we are more than just a software solutions provider – we are innovators, problem-solvers, and partners dedicated to driving success for businesses in a rapidly evolving digital landscape

VCODEZ
INNOVATING IDEAS

+91 9600189201
info@vcodez.com