Java Interview Questions :

WORA (Write Once Run Anywhere)

Java Features :

Simple: Java is quite simple to understand and the syntax
Platform Independent: Java is platform independent means we can run the same program in any software and hardware and will get the same result.
Interpreted: Java is interpreted as well as a compiler-based language.
Robust: features like Garbage collection, exception handling, etc that make the language robust.
Object-Oriented: Java is an object-oriented language that supports the concepts of class, objects, four pillars of OOPS, etc.
Secured: As we can directly share an application with the user without sharing the actual program makes Java a secure language.
High Performance: faster than other traditional interpreted programming languages.
Dynamic: supports dynamic loading of classes and interfaces.
Distributed: feature of Java makes us able to access files by calling the methods from any machine connected.
Multithreaded: deal with multiple tasks at once by defining multiple threads
Architecture Neutral: it is not dependent on the architecture.

Question :

Interviewer Question: "Why is Java called a platform-independent language?"

Your Answer:

Java is called **platform-independent** because the code we write is **not directly converted into machine-specific instructions.** Instead, when we compile a Java program using `javac`, it is converted into **bytecode** – a `.class` file.

This bytecode is **not specific to any one operating system or hardware.** It can run on any system that has a **JVM (Java Virtual Machine)** installed. The JVM is the one responsible for converting the bytecode into machine code that's understood by the operating system.

While the **JVM itself is platform-dependent** (meaning there's a different JVM for Windows, Mac, Linux, etc.), the **Java bytecode is the same everywhere.** So, I can write a Java program on Windows, compile it into bytecode, and then run that bytecode on a Mac or Linux system – as long as they have the appropriate JVM installed.

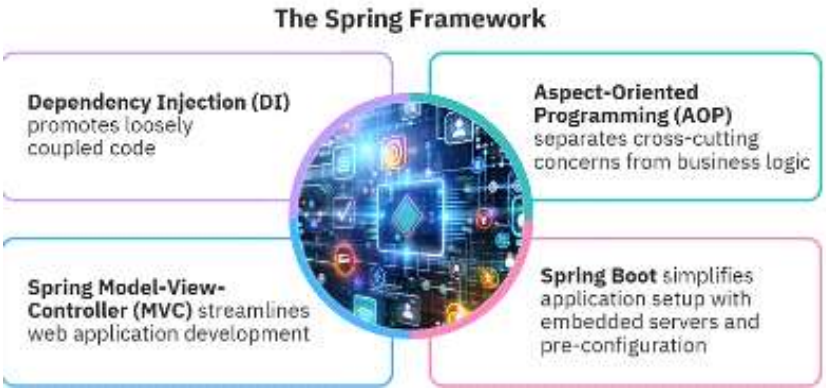
That's why we say **Java is platform-independent at the bytecode level.**

JAVA Ecosystem :

The Java ecosystem comprises various tools, libraries, frameworks, and platforms that allow developers to create robust, scalable, and secure applications.

Components of Java Ecosystem are :

- **Java Development Kit (JDK)**: This is the primary toolkit for Java developers, which includes:
  - **Java Compiler**: Converts Java source code into bytecode.
  - **Java Runtime Environment (JRE)**: Provides the necessary environment to run Java applications.
  - **Utilities**: Command-line tools that assist in the development process.
- **Integrated Development Environments (IDEs)**: Software applications that provide comprehensive facilities for software development. Popular IDEs include:
  - **Eclipse**
  - **IntelliJ IDEA**
  - **NetBeans**
- **Frameworks**: These facilitate code reuse and speed up development. Notable frameworks include:
  - **Spring Framework**: For enterprise applications.



- **Hibernate**: For object-relational mapping.

- **Build Tools**: Automate the process of compiling code and managing dependencies. Examples include:
  - **Maven**
  - **Gradle**
- **Application Servers**: Provide environments for deploying and managing Java applications. Examples include:
  - **Apache Tomcat**
  - **JBoss**
- **Testing Frameworks**: Ensure application functionality through testing. Common frameworks include:
  - **JUnit**
  - **TestNG**
- **Cloud and Microservices Support**: Tools for building cloud-native applications, such as **Spring Cloud**.

These components work together to create a powerful environment for developing Java applications. If you have any specific component you'd like to know more about, feel free to ask!

And if you want to continue exploring this topic, try one of these follow-up questions:

Note :

Three main components of Java . Interview Question

JVM

JRE

JDK

JVM : Java Virtual Machine : (It is a part of JRE)

JVM stands for Java Virtual Machine it is a Java interpreter(in simple it is a Translator) helps your computer understand Java programs.

The JVM takes this bytecode (code which is formed after compiled) and converts it into machine code, which is the language that your computer can understand and execute.

- This process allows Java programs to run on any device (which makes java aa Platform Independent). [this process is called JIT].

Note :

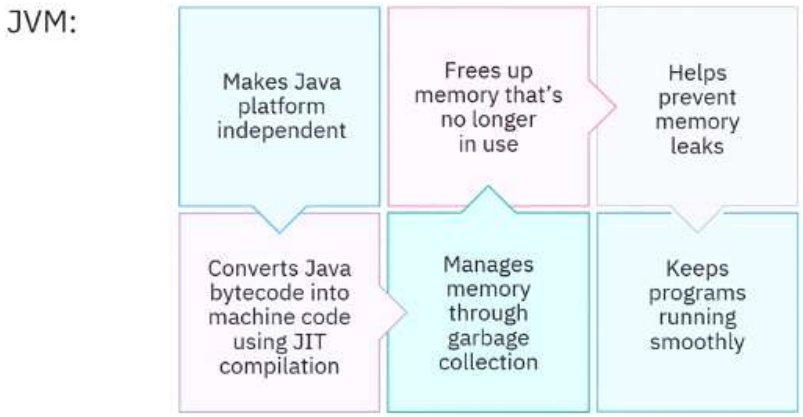
- It is responsible for loading, and executing the bytecode created in Java

As JVM is **platform dependent** [which means the software of JVM is different for different Operating Systems] it plays a vital role in making Java platform Independent.

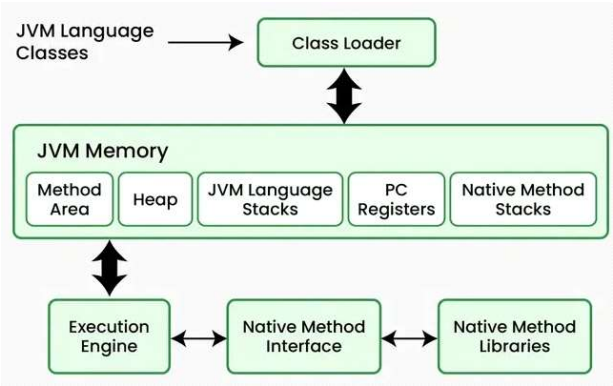
Means : Each OS has different JVM software's installed in it .

- Java applications are called **WORA (Write Once Run Anywhere)**. This means a programmer can develop Java code on one system and expect it to run on any other Java-enabled system without any adjustments. This is all possible because of the JVM.
- When we compile a **.java file**, and generated **.class files** (containing byte-code) this **.class file** goes through various steps when we run it. These steps together describe the whole JVM.

Features of JVM :



JVM Architecture:



Components :

1. Class Loader Subsystem

Class Loader Subsystem is a part of the JVM responsible for loading .class file and also for performing 3 main activities.

- Loading
- Linking
- Initialization

a. Loading: The Class loader reads the "*.class"* file, generate the binary data file, and save it in the **method area**(area part of JVM Memory).

For each "*.class"* file, JVM stores the following information in the method area.

- After loading the ".class" file, JVM creates an object of type Class to represent this file in the **heap memory.**

Note :

Please note that this object is of type lass predefined in **java.lang** package.

- These Class object can be used by the programmer for getting class level information like the name of the class, parent name, methods and variable information etc.

b. Linking: Performs verification, preparation, and (optionally) resolution.

Verification: It ensures the correctness of the *.class* file i.e. it checks whether this file is properly formatted and generated by a valid compiler or not.

- If verification fails, we get run-time exception ***java.lang.VerifyError*.**
- And this verification activity is done by the component called **ByteCodeVerifier**. Once this activity is completed then the class file is ready for compilation.

Preparation: JVM allocates memory for class static variables and initializing the memory to default values.

c. Initialization: In this phase, all static variables are assigned with their values defined in the code.

2. Class Loader :

When Java runs a program, it uses **class loaders** to load .class files into memory.

Means : **Class Loaders** are like "managers" that load Java classes into memory **when your program runs**. Java doesn't load all classes at once — it loads them when they are needed.

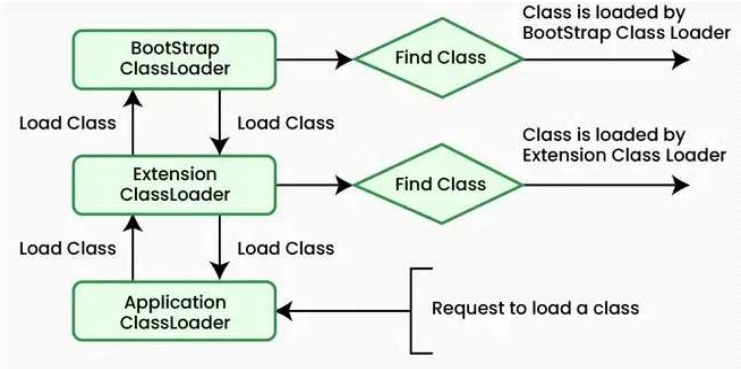
Java has three main types of Class Loaders:

- **Bootstrap Class Loader:**
- Loads **core Java classes** (like java.lang, java.util, etc.) present in the **"JAVA_HOME/lib"** directory.
- Example: `String.class`, `Object.class`
- **Extension (Platform) Class Loader**

  It is a child of the bootstrap class loader. It loads the classes present in the extensions directories "JAVA_HOME/jre/lib/ext"(Extension path) or any other directory specified by the java.ext.dirs system property.

- **Application (System) Class Loader**

  It is a child of the extension class loader. It is responsible to load classes from the application classpath. It internally uses Environment Variable which mapped to java.class.path.



Example:

```
public class Test {
    public static void main(String[] args) {
        System.out.println(String.class.getClassLoader());
        System.out.println(Test.class.getClassLoader());
    }
}
```

Note : **Difference between Class Loader Subsystem and Class Loader ?**

Inside this subsystem, Java uses different Class Loaders — like the Bootstrap, Extension, and Application class loaders — to load classes from various sources.

So, the Class Loader is an object that performs the loading, while the Class Loader Subsystem is the broader system that manages the entire process of preparing a class for execution."

3: **JVM Memory Areas**

JVM memory area refers to the memory management system employed by the Java Virtual Machine (JVM) to allocate and manage memory resources for running Java applications.

The JVM memory is broadly divided into several key areas:

- **Method area:** In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables.
- **Heap Area :** Heap Area stores Information of all objects. The Heap is managed by the JVM's Garbage Collector, which automatically reclaims memory occupied by objects that are no longer referenced.
- **Stack Area :** The Stack Area (also called JVM Stack), that stores method calls and local variables for each thread.

  In simple words :

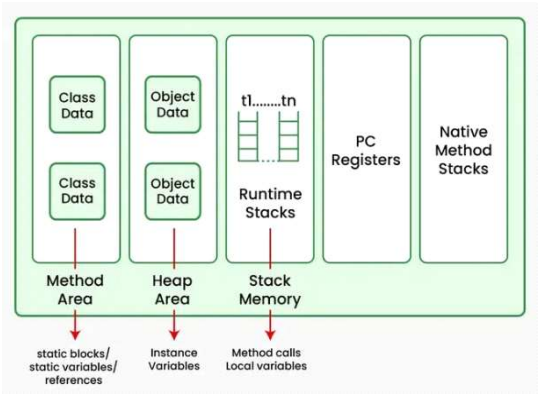
  Every time a method is called, a "stack frame" is created in the stack area.

  This stack frame stores:

- **Method arguments**
- **Local variables**
- **Intermediate results**
- **Return address** (where to go after the method finishes)

  When the method finishes, the frame is **popped (removed)** from the stack.

- **PC Registers:** Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
- **Native method stacks:** For every thread, a separate native stack is created. It stores native method information.



4. Execution Engine

Execution engine executes the ".class" (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:

- **Interpreter:** It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- **Just-In-Time Compiler(JIT):** It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to machine code, so whenever the interpreter sees **repeated method calls**, JIT provides direct machine code for that method call, so re-interpretation is not required, thus **efficiency is improved.**
- **Garbage Collector:** Garbage collection in Java is an automatic automatically **identifies and removes unused objects**, freeing up memory in the heap. (objects and there memory which are not used for longer period of time are freed up).

  Java programs compile to bytecode that can be run on a Java Virtual Machine (JVM). When Java programs run on the JVM, objects in the heap are created, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

5. Java Native Interface (JNI)

It is an interface that interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

6. Native Method Libraries

These are collections of native libraries required for executing native methods. They include libraries written in languages like C and C++.

🔍 Difference Between Heap and Stack Memory in Java

Feature	Stack Memory	Heap Memory
Used for	Storing **method calls**, **local variables**, and **references**	Storing **objects** and **instance variables**
Scope	Per thread (each thread has its own stack)	Shared across all threads
Access	Fast, because it's small and managed with LIFO	Slower than stack, because it's larger and uses dynamic allocation
Lifespan	Exists **only while method is running**	Exists **until no references remain (GC)**
Managed by	JVM internally	JVM + **Garbage Collector**
Error	`StackOverflowError` (e.g., deep recursion)	`OutOfMemoryError` (e.g., too many objects)

Example :

```
public class Test {
    public static void main(String[] args) {
        int x = 10;                 // x is in stack
        Person p = new Person();    // 'p' is in stack, new Person() is in heap
    }
}

class Person {
    String name;   // Instance variables go in heap
}
```

▶️ Memory View:

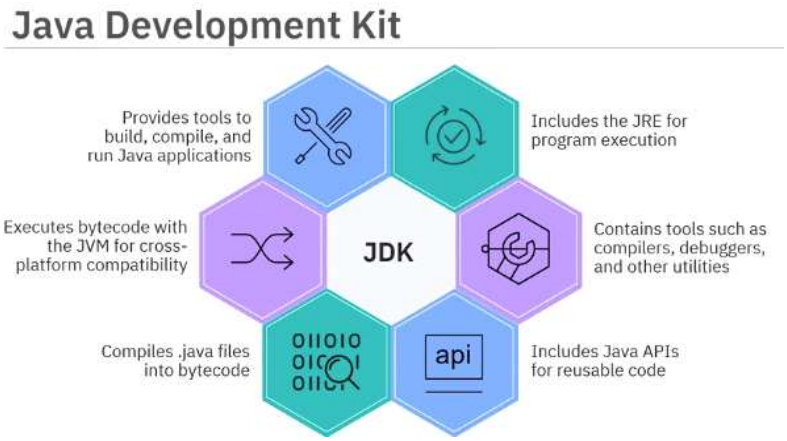
- `x = 10` is stored **in the stack**.
- `p` (the reference to the object) is **in the stack**.
- `new Person()` (the actual object) and its `name` field are **in the heap**.

✅ How to Answer in an Interview

> "In Java, Stack and Heap are two key memory areas. Stack is used for method execution and local variables, and it's created per thread. Heap is used to store objects and is shared across all threads. For example, when we declare an `int` variable in a method, it's stored in the stack. But if we create an object using `new`, the object goes to the heap, while its reference stays in the stack. Java uses garbage collection to clean up unused objects from the heap, but the stack memory is cleared automatically when a method finishes."

JDK : Java Development kit

The Java Development Kit (JDK) is essential for Java development as it provides all the necessary tools and components to create, compile, and run Java applications.



Here's how it helps:

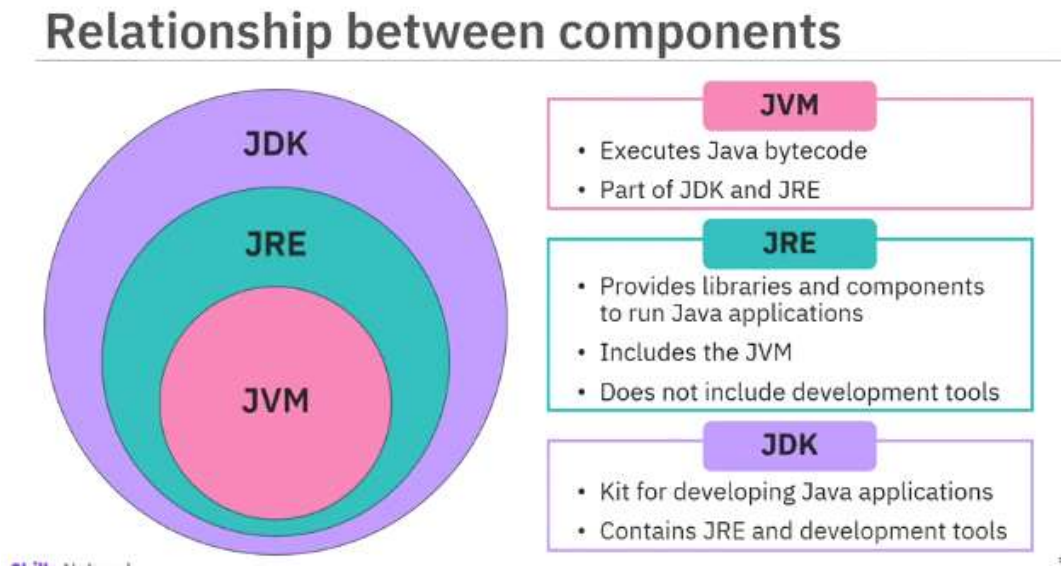
- **Complete Toolkit**: The JDK includes everything you need to develop Java applications, such as the Java Runtime Environment (JRE), compilers, and debugging tools.
- **Compilation**: When you write a Java program, you save it as a .java file. The JDK contains the javac compiler, which converts this .java file into bytecode (a .class file) that the JVM can execute.
- **Development Tools**: The JDK provides various utilities, such as debuggers and Java APIs (libraries of reusable code), which help streamline the development process and make coding easier.
- **Environment Setup**: It helps set up the Java environment on your computer, ensuring that you have everything configured correctly to start programming in Java.

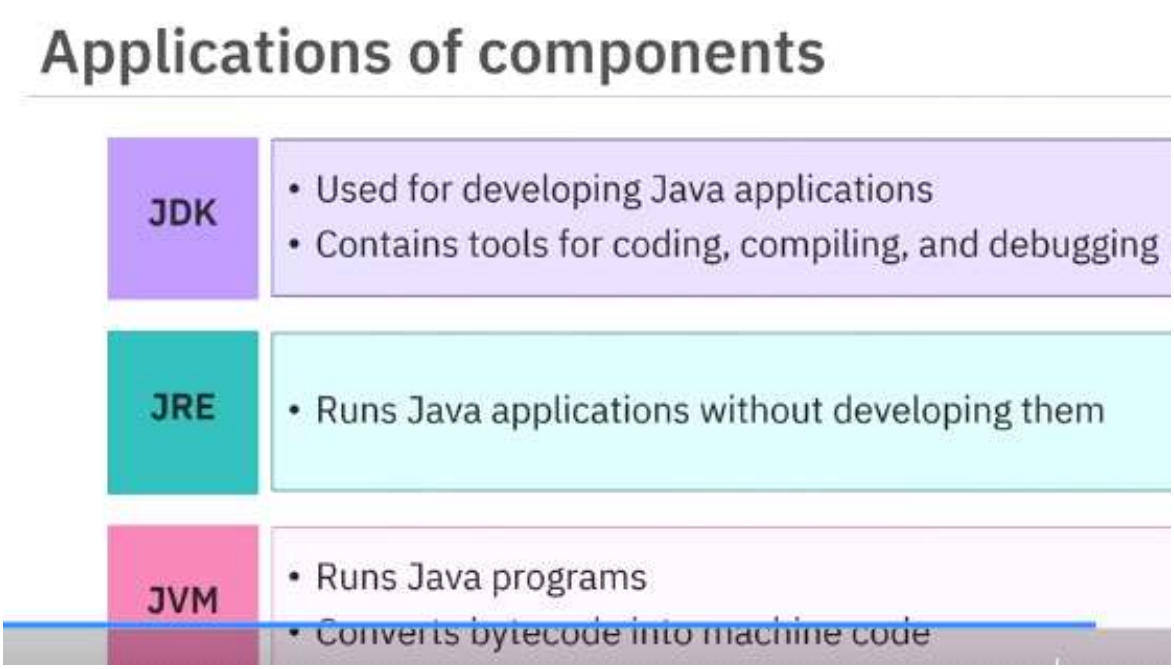
JRE : Java Runtime Environment

The Java Runtime Environment (JRE) is a crucial component for running Java applications. Here's a simple breakdown of its role:

- **Execution Environment**: The JRE provides the environment needed to execute Java programs. It includes the Java Virtual Machine (JVM), which is responsible for running the Java bytecode.
- **Core Libraries**: The JRE contains essential libraries and components that Java applications need to function. These libraries provide pre-written code for common tasks, such as input/output operations and networking.

- **No Development Tools**: Unlike the Java Development Kit (JDK), the JRE does not include development tools like compilers. It is designed solely for running Java applications, not for creating them.
- **Lightweight**: The JRE is lighter than the JDK, making it suitable for users who only need to run Java applications without developing them.





Can java be said to be the complete object-oriented programming language?

Java is not a *pure* or *completely* object-oriented programming language, because it supports primitive data types like int, float, char, bool, etc., which are **not objects** (or which are not Integer Class which is a Wrapper Class).

Detailed :

🧠 Detailed Explanation:

✅ **Java is Object-Oriented because:**

- It uses **classes and objects**.
- Supports all **OOP principles**:
  - **Encapsulation**
  - **Abstraction**
  - **Inheritance**
  - **Polymorphism**
- Almost everything in Java is part of a **class** or object.
- You can build large systems using only **object-oriented** design.

❌ **Java is not fully Object-Oriented because:**

- It supports **primitive types** (`int` , `boolean` , `char` , etc.) that are **not objects**.
- These are included for **performance reasons**.
- Unlike **pure OO languages** (like Smalltalk), not *everything* in Java is an object.

Note :

Wrapper Class :

Java introduced **wrapper classes** (`Integer` , `Double` , `Boolean` , etc.) so that primitives can be treated like objects when needed (e.g., in collections). This helps **bridge the gap** but doesn't make Java fully OO.

- Java Collections Framework (e.g., ArrayList<Integer>, HashMap<Integer, String>)

How is Java different from C++?

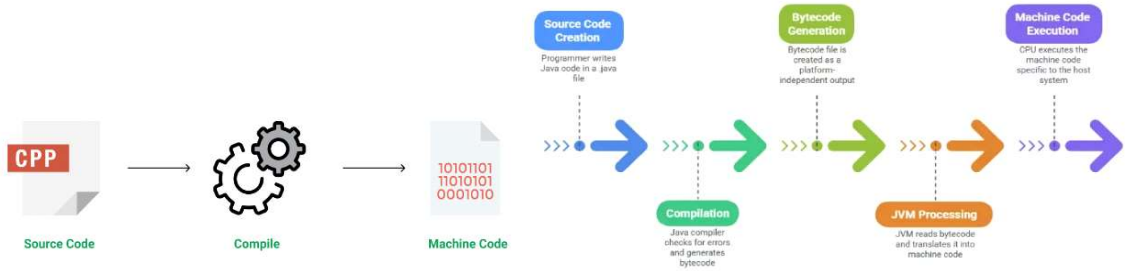
C++	JAVA

C++ Programs are machine dependent, they can only run in machine which it is compiled .	Java programs are machine-independent, it follows WORA
C++ is only a compiled language	Java is compiled as well as an interpreted language.
C++ allows pointers in the program.	Where java does not allows pointers.
C++ supports the concept of **Multiple inheritances**	Java does not supports it, to support it we use Interface .
Memory Management in C++ is Manual, using Constructor and Destructors.	Memory Management is System Controlled using **Garbage Collector.**
C++ supports **Structures** and **Unions**.	Java doesn't support Structures and Unions.
C++ allows Virtual Keyword	Java does not allow Virtual Keyword
It supports both method and operator overloading.	It supports only method overloading, not operator overloading.
C++ is used for developing the System Hardware	Java is used for developing the Application programs
C++ supports using Global Variables	Java does not support using Global Variabbles.

Performance (or) Compilation :

Performance:

- **C++** is generally **faster** because it compiles to native machine code and allows direct hardware access. It's often used for performance-critical applications like game engines, operating systems, and high-frequency trading software.
- **Java** can be slightly **slower** due to the overhead of the JVM and the bytecode interpretation. However, modern JVMs with Just-In-Time (JIT) compilers have significantly closed the performance gap.



Note :

Explain why C++ is both procedural and object oriented programming language?

C++ is considered a multi-paradigm programming language because it supports both procedural and

object-oriented programming paradigms.

Procedural Aspects: it supports

1 : Functions

2: Global Variables

3: Direct Memory Access (using Pointers)

Object-Oriented Aspects: it supports

1: Classes and Objects

2: Encapsulation

3: Inheritance

4: Abstraction, etc..

How C++ allocates and de-allocates the memory manually ?

In C++, memory management is manually performed by “manual memory management using constructors and destructors”.

- It allocates memory using “new” keyword, and deletes memory using “delete” keyword.

Why Java does not use pointers?

One of Java’s main goals is to be:

- **Secure**: No direct memory access = fewer bugs & attacks.
- **Simple**: Easier to learn and maintain code without pointer-related bugs.

Java does not use pointers same like C++ because, it does not provide direct memory access.

- This design makes Java enhances security, simplifies memory management.

Instead of traditional pointers, **Java utilizes references.**

- These references are like "safe pointers" as they point to objects in the heap but prevent direct manipulation of memory addresses.

Means : [references acts similar to pointers by **allowing access to objects** in an abstract way (the low-level memory details are hidden), making Java a safer.]

And,

The Java Virtual Machine (JVM) handles memory management, including garbage collection, which automatically reclaims memory occupied by unreferenced objects. This automatic process eliminates common pointer-related issues like dangling pointers and memory leaks.

Disadvantage of Pointers:

- Due, to usage of pointers there can be a serious issues like memory corruption, buffer

overflows, and some security issues.

Use of Virtual Keyword ?

The virtual keyword in C++ is primarily used to enable runtime polymorphism **through virtual functions.**

Enabling Polymorphism:

When a member function in a base class is declared `virtual`, it signals to the compiler that this function can be overridden by derived classes. This allows for dynamic dispatch, meaning the specific implementation of the function to be called is determined at runtime based on the actual type of the object pointed to by a base class pointer or reference.

In essence, the `virtual` keyword facilitates flexible and extensible class hierarchies by allowing different behaviors for objects of derived classes to be invoked through a common base class interface at runtime.

What is Pure Object Oriented Programming Language ?

A programming language is considered "purely object-oriented" when it strictly follows the **object-oriented paradigm**, **treating everything** within the language as an object.

- This includes not only user-defined classes and their instances but also primitive data types like integers, characters, and Booleans should be treated as object.

Key characteristics that define a purely object-oriented language include:

1 : Everything is an Object

2 : Message Passing : object is passed message before performing the operation.

3: Encapsulation

4: Abstraction

5: Inheritance

6: Polymorphism

Which is Pure Object Oriented Programming Language ?

Smalltalk is widely recognized as the quintessential example of a pure object-oriented programming language. In Smalltalk, all data types, even numbers and booleans, are instances of classes and behave as objects.

C++ and JAVA which is pure object oriented programming language?

Java is generally considered to be closer to a "pure" object-oriented programming language than C++.

Reasons for Java's "Purity":

- **Everything is an object (mostly):**

In Java, almost everything, including the main method, must reside within a class.

Primitive data types (like int, float, boolean) are the primary exception,but though they have corresponding wrapper classes (e.g., Integer, Float, Boolean) that are objects.

- No Global Variables
- Inheritance

Structure of Java :

The structure of Java code is organized in a way that promotes clarity and maintainability. Here are the key components:

- **Comments**:
  - Used to explain code and improve readability.
  - Three types of comments:
    - **Single-line**: // This is a comment
    - **Multi-line**: /* This is a multi-line comment */
    - **Documentation**: /** This is a documentation comment */
- **Packages**:
  - Used to group related classes and interfaces.
  - Declared at the top of the Java source file using the package keyword.

- **Classes**:
  - A class is a blueprint for creating objects and contains methods and variables.
  - Each public class should be in its own source file named after the class with a .java extension.

- Example:
- public class MyClass {
- // Class content

}

- **Main Method**:
  - For every Java application there should be an entry point where the java application starts running , typically the main method.
  - It is defined as:
  - public static void main(String[] args) {
  - // Code to execute

}

Note : Interview Question

11. What will happen if we don't declare the main as static?

We can declare the main method without using static and without getting any errors. But, the main method will not be treated as the entry point to the application or the program.

- **Methods**:
  - Define behaviors and functionalities within a class.
  - Should have descriptive names and follow naming conventions (e.g., camelCase).
  - Example:
  - public void myMethod() {
  - // Method content

}

- **Variables**:
  - Used to store data.
  - Must be declared with a specific data type.
  - Example:

int number = 10;

- **Import Statements**:
  - Used to include classes from other packages.

- Declared at the top of the file.
- Example: import java.util.List; // importing classes from the Java Standard Library.

Example

```
import java.util.List;
import java.util.ArrayList; // Importing classes from Java's standard
library

package com.example.library;

public class Library {
    // Class code
}
```

Note :

Source File and Naming Convention :

If you have a class name book then the file name should of book. Java

Source files and naming conventions

- Each public class should be in its own source file
- Named after the class with a .java extension

Example

```
// Book.java
package com.example.library;

public class Book {
    String title;
    String author;

    public Book(String title, String
author) {
        this.title = title;
        this.author = author;
    }
}
```

Java Data Types :

Difference between instance variable and a local variable?

Instance Variable : Instance variables are those variables that are accessible by all the methods in the class. They are declared outside the methods and inside the class. These variables describe the properties of an object and remain bound to it at any cost.

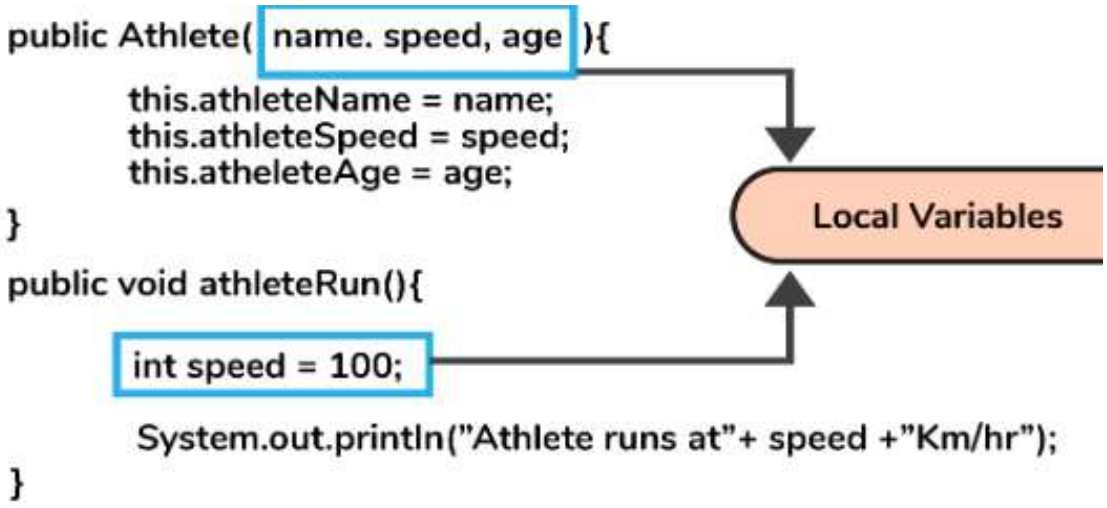
Example:

```
class Athlete {
public String athleteName;
public double athleteSpeed;
public int athleteAge;
}
```

Local variables: Local variables are those variables present within a block, function, or constructor and can be accessed only inside them.

The utilization of the variable is restricted to the block scope.

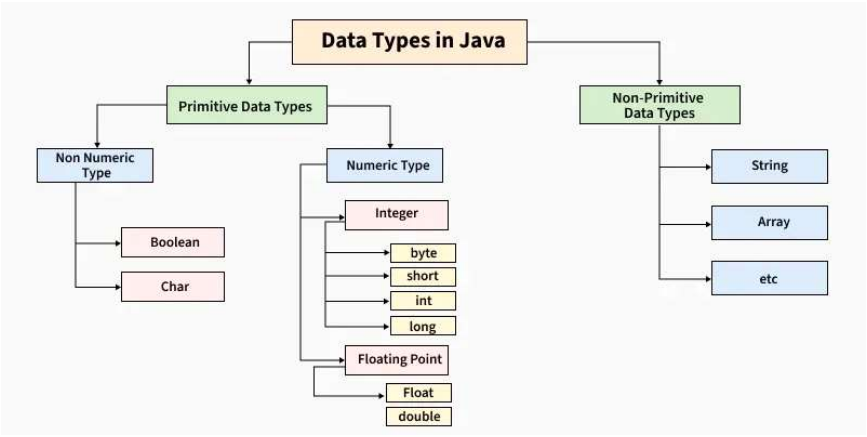
Whenever a local variable is declared inside a method, the other class methods don't have any knowledge about the local variable.



JAVA Data Types :

Java is statically typed programming language which means variable types are known at the compile time.

- In Java, compiler knows exactly what types each variable holds and enforces correct usage during compilation. For example "int x = "GfG";" gives compiler error in Java as we are trying to store string in an integer type.



Data Types refers to the type of data that a variable can store.

Data Types also defines as “Containers” to hold the specific type of data .

Data Types have a specific properties called : Size, Type, and Range.

In Java, data types are categorized into two main groups: **primitive data types** and **reference data types**. Here's a brief overview of each:

- **Primitive Data Types**

These are the basic data types that hold simple values.

- They have a fixed size and cannot contain null values.

The main primitive data types in Java are:

Primitive data types in Java

Data Type	Use when ...	Usage example	Code example
Byte	You need to save memory in large arrays with values between -128 and 127	Storing age or small numerical values	`byte age = 25; // Age of a person`
Short	You need a small integer value and want to save memory, but the values are between -32,768 and 32,767	Temperature values in Celsius	`short temperature = -5; // Temperature in degrees`
Int	You need to store whole numbers larger than what byte and short can hold	Population counts or counters	`int population = 1000000; // Population of a city`

Data Type	Use when ...	Usage example	Code example
Long	You need to store very large integer values that exceed the range of **int**	Distances or timestamps	`long distanceToMoon = 384400000L; // Distance in meters`
Float	You need to store decimal numbers that do not require high precision – up to seven decimal places	Prices of products	`float price = 19.99f; // Prices of products`
Double	You need to store decimal numbers requiring high precision – up to 15 decimal places	Scientific calculations or precise measurements	`double pi = 3.141592653589793; // Value of Pi`

Data Type	Use when ...	Usage example	Code example
Char	You need to store a single character	Initials or single-letter codes	`char initial = 'A'; // Initial of a person's name`
Boolean	You must represent a true or false value, which is often used for conditions and decisions	Checking if a user is logged in	`boolean isLoggedIn = true; // User login status`

Default Values for each type :

boolean	false
byte	0
char	\u0000
short	0
int	0
long	0L
float	0.0f
double	0.0d

- **Reference Data Types**

These data types refer to objects and can hold complex data structures. They include:

1. Strings

Strings are defined as an array of characters.

The difference between a **character array** and a **string** in Java is, that the **string** is designed to hold a sequence of characters in a single variable whereas, a **character** array is a collection of separate char-type entities.

Reference Data Type	Characteristics	Useful for	Example
Strings	• Consist of any sequence or combination of text or characters • Useful for handling text	Handling data	`String greeting = "Hello, World!";`
Arrays	• Store multiple values of the same type • Useful for storing lists of items	Storing lists of items	`int[] scores = {85, 90, 78, 92};`

2: Class

Class in programming is like a blueprint or template for creating objects (instances). It defines properties (attributes) and behaviours (methods) that the objects created from the class will have, and these properties are common to all objects of one type.

Reference Data Type	Characteristics	Useful for	Examples

Classes	• Are similar to blueprints
	• Used to create objects
		• Organizing related data and functions together
		• Creating templates for multiple similar items
		• Enforcing consistent structure and behavior
			`class Car {`
			`  String color;`
			`  int year;`
			`  void displayInfo() {`
			`  System.out.println("Color: " + color + ", Year: " + year);`
			`  }`
			`}`

3: Object

An Object is an instance of Class, which is represents real-world entity.

An object is built from a blueprint of class, like : An object is like a specific house built from that blueprint.

An object consists of :

- **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Reference Data Type	Characteristics	Useful for	Examples
Objects	• Are classes that contain both data and functions • Objects operate on the data	• Storing specific instances of data • Performing data operations • Representing real-world entities in code	`public class Main { public static void main(String[] args) { Car myCar = new Car(); myCar.color = "Red"; myCar.year = 2026; myCar.displayInfo(); // Output: Color: Red, Year: 2026 } }`

4: Interface (blueprint of a behaviour)

An **Interface in Java** programming language is defined as an abstract type used to specify the behaviour of a class.

Interfaces specify what a class must do and not how. It is the blueprint of the class.

Key Properties :

- **The interface in Java is a mechanism to achieve abstraction.**
- Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

Example :

```
interface Add{
   int add(int a,int b);
}
```

- By default, **variables in an interface are public, static, and final.**
- It is used to achieve abstraction and multiple inheritance in Java.

Reference Data Type	Characteristics	Useful for	Example

Interfaces	• Define required methods for implementing classes
	• Only declare the methods without providing actual code
		• Provide a blueprint for classes
		• Enable a class to use features from multiple sources
		• Enable the use of multiple classes from one interface
		• Ease maintenance
			`interface MyInterfaceClass {`
			`  void methodExampleOne();`
			`  void methodExampleTwo();`
			`  void methodExampleThree();`

Features	Class	Interface
Instantiation	In class, you can create an object.	In an interface, you can't create an object
Variables	Class can have instance variables	Variables are public static final (constants only).
Methods	Class can have concrete methods	In an interface, methods are abstract by default
Inheritance	It supports single inheritance	Supports multiple inheritance
Constructors	Can have constructors.	No constructors allowed.
Access Modifiers	Supports private, protected, public, default.	In an interface, all members are public by default
Keyword	Defined using class.	Defined using interface.

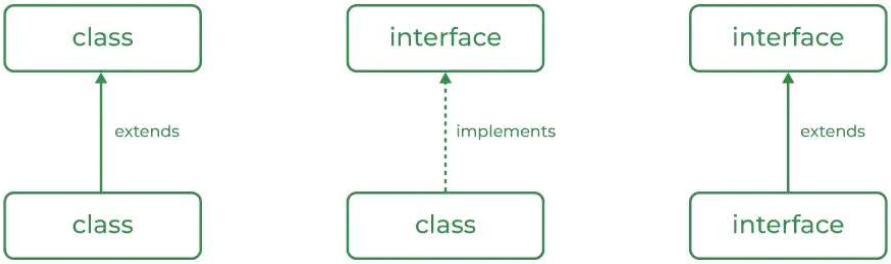
When to Use Class and Interface?

- **Use a Class when**:
    - Use a class when you need to represent a real-world entity with attributes (fields) and behaviors (methods).
    - Use a class when you need to create objects that hold state and identity to perform actions.
- **Use a Interface when:**
    - Use an interface when you need to define a behavior that multiple classes can implement.
    - Interface is ideal for achieving abstraction and multiple inheritance.

Reference Data Type	Characteristics	Useful for	Example

Enums	• Represent fixed sets of named values	Representing fixed sets of options	}
			`enum DaysOfWeek {`
			`MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;`
			}

Note :



- Class **extends** a Class.
- Interface **extends** a Interface .
- Class **implements** a Interface.

Note :

```java
// Public interface accessible anywhere
interface Display {

    // Public method accessible anywhere
    public void showPublic();


    // Default method (introduced in Java 8)
    // is accessible to any class that implements the interface,
    // regardless of the package.
    default void showDefault() {
        System.out.println("Method in the interface.");
    }
}
```

Difference between Struct, Enum, Union in C++ ? (Interview)

Structure

In C++, **structure** is a user-defined data type that is used to combine data of different types.

It is similar to an array but unlike an array, which stores elements of the same type, a structure can store elements of different data types.

- C++ structures can also have member functions to manipulate its data.

Syntax:

```
struct name{
    type1 mem1;
    type2 mem2;
    ...
};
```

Example :

```
struct GfG {
    char c;
    int x, y;
};
```

Size of Structure

The size of a structure is determined by the sum of the sizes of its individual data members. For above example, size = 1 + 8 = 9.

Note :

But in some situation it also add additional padding added by the compiler to ensure proper memory alignment.

typedef

In C++, typedef is used to create an alias for an existing variable. Similarly, with structures, typedef creates an alias for the original name of the structure.

```
typedef struct GeeksforGeeks {
    int x, y;

// Alias is specified here
} GfG;
```

Union

In C++, **union** is a user-defined datatype in which we can define members of different types of data types just like structures but unlike a structure, where each member has its own memory, a **union** member shares the same memory location.

Syntax :

```
union union_name{
    type1 member1;
    type2 member2;
    .
    .
    typeN memberN;
};
```

Example :

```
union geek {
    int age;
    float GPA;
    double marks;
};
```

O/P : 8

Size of Union

All the elements of the union are stored in the same memory location. So, if the size of the union is equal to the **size of the largest element.**

Enumeration

In C++, **enumeration** (enum) is a user-defined type that consists of a set of named integral constants.

- Enumerations help to assign assigning meaningful names to integral constants that makes the code more readable and easier to maintain.

Syntax :

```
enum enum_name {
    name1, name2, name3, ...
};
```

Example :

```
enum fruit {
    APPLE, BANANA = 5, ORANGE
};
```

By default, the **first name** in an enum is assigned the integer value 0, and the subsequent ones are incremented by 1.

Java Operators :

Operators are special symbols that perform operations on variables and values. They are essential for manipulating data and can be categorized into several types:

Arithmetic operators

- Perform basic arithmetic operations

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus	a % b

Relational operators

- Compare two values
- Return a Boolean result (true or false)

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

Logical operators

- Combine multiple Boolean expressions

Operator	Description	Example
&&	Logical AND	(a > b) && (b < c)
!	Logical NOT	!(a > b)

Advanced operators in Java

Go beyond basic arithmetic, comparison, and logical operators

Bitwise operators: Binary data

Used for
- Low-level programming
- Graphics
- Cryptography

Shift operators: Shifting bits in numbers

Assignment operators

Assign values to variables

Operator	Description	Example
=	Assigns right operand to left	a = 10
+=	Adds and assigns	a += 5
-=	Subtracts and assigns	a -= 2
*=	Multiplies and assigns	a *= 3
/=	Divides and assigns	a /= 2
%=	Takes modulus and assigns	a %= 4

Unary operators

- Operate on a single operand
- Used for incrementing, decrementing, or negating values

Operator	Description	Example
+	Unary plus	+a
-	Unary minus	-a
++	Increment	++a or a++

Ternary operators

Shorthand form of the conditional statement

Takes three operands

Syntax

```
condition ? expression1 : expression2;
```

Arrays in Java :

What is an array in Java?

Declaring an array

Use the following syntax:

1. Specify the data type followed by an opening and closing bracket
2. Type a space
3. Specify the array name
4. Type a semicolon

```
dataType[] arrayName;
```

Initializing an array

- Allocate memory to initialize the array
- Add the keyword **new**

```
numbers = new int[5];
```

Initializing an array

- Declare and initialize an array in a single line

```
int[] numbers = new int[5];
```

Initializing an array

1. Create an array
2. Assign values using curly braces

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

Iterating through arrays

- Used to read or modify elements

```
for (int i = 0;
i < numbers.length; i++)
 {
 System.out.println(numbers[i]);
```

- Use an enhanced **for** loop

```
for (int number : numbers)
{
 System.out.println(number);
}
```

Working with multi-dimensional arrays

- Are arrays of arrays
- Two-dimensional arrays are the most common

```
int[][] matrix = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};
```

- Double square brackets indicate a 2D array
- Numbers within curly braces are the values
- Each set of curly braces represents a row

```
int[][] matrix = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};
```

Iterating through a 2D array

- The first **for** loop creates the outer loop
- The second **for** loop handles each column in the current row
- The **system out** message prints each number in the grid
- The **system out print-in** message moves the output to a new line

```
for (int i = 0; i < matrix.length; i++) {
  for (int j = 0; j < matrix[i].length; j++) {
    System.out.print(matrix[i][j] + " ");
  }
  System.out.println(); // Move to the next line after each row
}
```

Conditional Statements in Java :

The 'if' statement

- States that the integer number equals 10
- Checks if the number is greater than five
- Concludes that the number 10 is greater than the number five
- Prints the specified statement

```
public class Main {
    public static void main(String[] args) {
        int number = 10;

        if (number > 5) {
            System.out.println("The number is greater than 5.");
        }
    }
}
```

The 'if-else' statement

- Checks for a specific condition
- If the condition is false, the program runs the code

```
public class Main {
    public static void main(String[]
args) {
        int number = 3;
        if (number > 5) {
            System.out.println("The
number is greater than 5.");
        } else {
            System.out.println("The
number is not greater than 5.");
        }
    }
}
```

The 'else-if' statement

Checks for multiple conditions

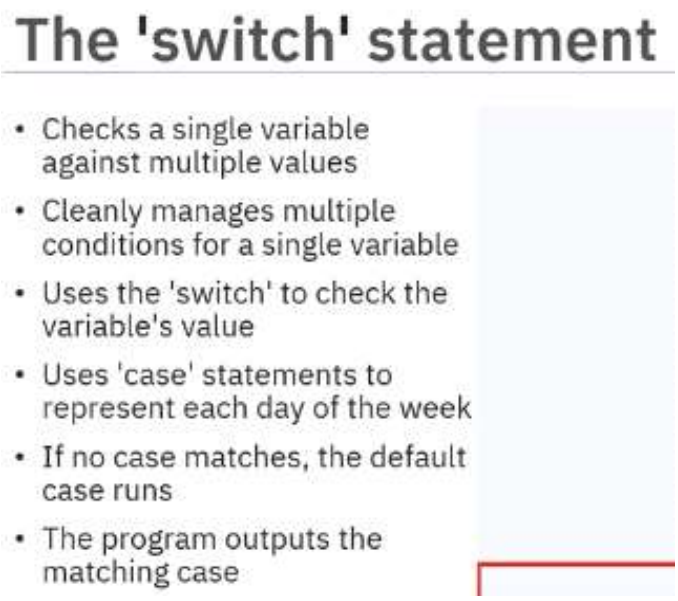
- Has an integer variable of 5
- The first 'if' condition prints if the number is greater than five.
- The 'else if' condition prints if the number equals five.
- The 'else' condition prints if the number is less than five
- The integer meets the second condition

```
public class Main {
    public static void main(String[]
args) {
        int number = 5;
        if (number > 5) {
            System.out.println("The
number is greater than 5.");
        } else if (number == 5) {
            System.out.println("The
number is equal to 5.");
        } else {
            System.out.println("The
number is less than 5.");
        }
    }
}
```

Nested conditional statements

- Uses an if statement to check the first condition
- Nests a second if statement to check another condition
- Outputs a statement based on meeting the if or else condition

```
public class Main {
    public static void main(String[]
args) {
        int age = 20;
        if (age >= 18) {
            System.out.println
("You are an adult.");
            if (age >= 65) {
                System.out.println
("You are a senior citizen.");
            }
        } else {
            System.out.println
("You are a minor.");
        }
```



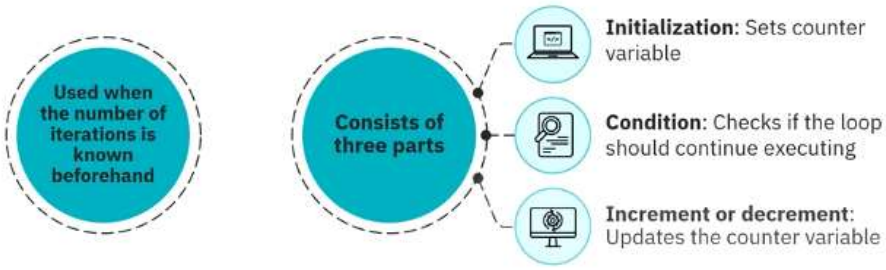
Decision Making in Java

Statement	Use Case	Example
if	Single condition check	if (age >= 18)
if-else	Two-way decision	if (x > y) {...} else {...}
nested-if	Multi-level conditions	if (x > 10) { if (y > 5) {...} }
if-else-if	Multiple conditions	if (marks >= 90) {...} else if (marks >= 80) {...}
switch-case	Exact value matching	switch (day) { case 1: ... }
break	Exit loop/switch	break;
continue	Skip iteration	continue;
return	Exit method	return result;

Loops :

Types of loops: For loop

Definition

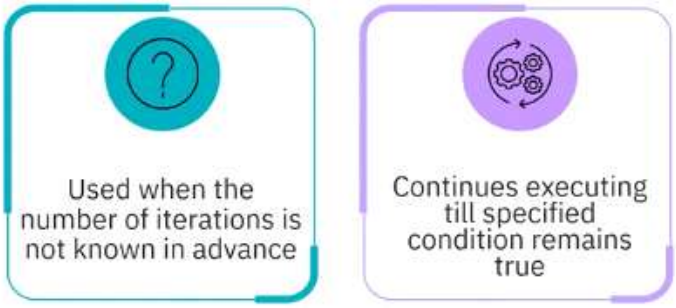


Syntax

```
for (initialization; condition; increment/decrement) {

    // Code to be executed

}
```

Types of loops: While loop

Definition



Syntax

```
while (condition) {

    // Code to be executed

}
```

Example :

```
int i = 1; // Initialization
while (i <= 5) { // Condition
    System.out.println(i);
    i++; // Increment
}
```

Types of loops: Do-while loop

Definition

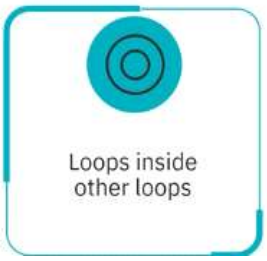
Syntax

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

Example :

```
    int i = 1; // Initialization
    do {
        System.out.println(i);
        i++; // Increment
    } while (i <= 5); // Condition
}
```

Nested loops



Example :

```
for (int i = 1; i <= 10; i++) { // Outer loop for rows
    for (int j = 1; j <= 10; j++) { // Inner loop for columns
        System.out.print(i * j + "\t"); // Print product with tab

    }
    System.out.println(); // Move to the next line after inner
completes
```

Break statement

Definition

Example :

```
        int[] numbers = {1, 3, 5, 7, 9, 2, 4};
        for (int num : numbers) {
            if (num > 5) {
                System.out.println("First number greater than 5: " +
num);
                break; // Exit the loop
            }
        }
```

Continue statement

Definition

Example :

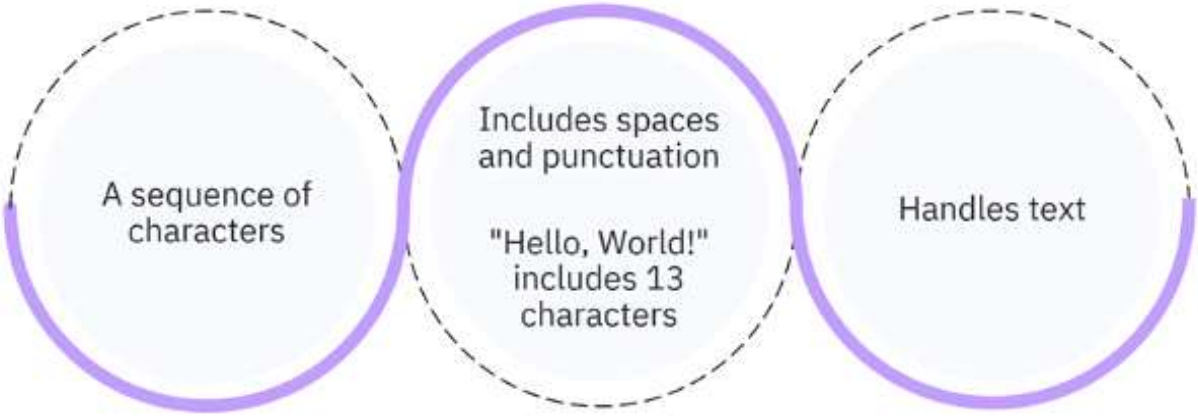
```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        continue; // Skip the rest of this iteration
    }
    System.out.println(i);
}
```

Loops :

Loop Type	When to Use	Condition Checking	Executes At Least Once?

for loop	When you want exact iterations	Before loop body, It is called Entry-controlled.	no
while loop	When you need condition check first.	Before loop body, It is called Entry-controlled.	no
do-while loop	When you need to run at least once	After loop body, It is called Exit-controlled.	yes
for-each loop	When you process all collection items	Internally handled	no

What are strings?



Creating a string

```
//String literals

String greeting = "Hello, World!";

//New keyword

String message = new String("Hello, World!");
```

- Strings are created in two ways:
  - Text is enclosed in double quotes
  - The keyword "new" is used
- Examples:
  - "Hello, World!" represents a string literal
  - new String("Hello World!") creates a string object

Counting string characters

```
String text = "Java Programming";

int length = text.length();

System.out.println("Length of the string: " + length); // Output: 18
```

- Method: length()
- Returns local character count
- Includes spaces
- Examples:
  - "Java Programming" has 18 characters

Accessing string characters

```
String word = "Java";

char firstChar = word.charAt(0);

System.out.println("First character: " + firstChar); // Output: J
```

- Method: charAt()
- Returns the character to a position
- Starts from index 0
- Example:
  - charAt(0) for the string "Java: returns J"

Concatenate strings

```java
//Using the + operator
String firstName = "John";
String lastName = "Doe";
String fullName = firstName + " " + lastName; //
Using +
System.out.println("Full Name: " + fullName); //
Output: John Doe

//Using the concat() method
String anotherFullName = firstName.concat("
").concat(lastName);
System.out.println("Another Full Name: " +
anotherFullName); // Output: John Doe
```

- Executed using:
  - "+" operator
  - concat() method
- Example:

Compare strings

```java
String str1 = "Hello";
String str2 = "Hello";
String str3 = "World";
boolean isEqual = str1.equals(str2);
System.out.println("str1 equals str2: " +
isEqual); // Output: true
boolean isNotEqual = str1.equals(str3);
System.out.println("str1 equals str3: " +
isNotEqual); // Output: false
```

- Method: equals()
- Checks of strings have identical content
- Ensures actual character match
- Example:
  - Same strings return true
  - Different strings return false

Extract substrings

```java
String phrase = "Java Programming";
String sub = phrase.substring(5, 16);
System.out.println("Substring: " + sub); // Output: Programming
```

- Method: substring()
- Allows specifying starting and ending index
- Example:
  - Consider the string "Java Programming"
  - Does not include ending indices

Split strings

```java
String csv = "apple,banana,cherry";
String[] fruits = csv.split(",");
for (String fruit : fruits) {
    System.out.println(fruit);
}
// Output:
// apple
// banana
// cherry
```

- Method: split()
- Breaks a string into smaller pieces
- Uses a delimiter
- Example:
  - Splitting at the comma results in separate elements

Join strings

```
String[] colors = {"Red", "Green", "Blue"};

String joinedColors = String.join(", ", colors);

System.out.println(joinedColors); // Output: Red, Green, Blue
```

- Method: String.join
- Combines array into a string
- Allows separator
- Example:
  - Uses a comma and space as a separator
  - Creates a new string

Difference between **concatenate string** and **join string** method in java?

1. `concatenate()` method

In Java, `concatenate()` is actually not a method in the `String` class. What people usually refer to is the `+` operator or `String.concat()` method.

- `String.concat()` is a method that appends one string to another.
- It is used to join two strings.

Example of using `String.concat()` :

java Copy Edit

```
String str1 = "Hello";
String str2 = "World";
String result = str1.concat(" ").concat(str2); // "Hello World"
```

Efficiency: Using concat() can **lead to inefficiencies** because **String is immutable** in Java, and each concatenation creates a new String object. If you're performing many concatenations, it can result in poor performance.

2. `join()` method

The `String.join()` method was introduced in Java 8 and is used to join multiple strings (or elements from an array or collection) with a specified delimiter.

Example of using `String.join()`:

java Copy Edit

```java
String[] words = {"Hello", "World"};
String result = String.join(" ", words); // "Hello World"
```

- **Delimiter:** The first argument is a delimiter that separates the strings. If you want no separator, you can pass an empty string (`""`).
- **Efficiency:** `String.join()` is typically more efficient than repeatedly using `concat()` or the `+` operator because it internally uses a `StringBuilder` to build the result.

Efficiency: String.join() is typically more efficient than repeatedly using concat() or the + operator because it internally uses a StringBuilder to build the result.

Note :

Delimiter: The first argument is a delimiter that separates the strings. If you want no separator, you can pass an empty string ("").

String immutability

```java
String original = "Hello";

original = original + " World"; // Creates a new string

System.out.println(original); // Output: Hello World
```

- Strings cannot be changed once created
- A new string is created when strings are modified
- Example:
  - A new string is created
  - Old string is left unaltered

Built-in methods

```
//toUpperCase()
String text = "hello";
System.out.println(text.toUpperCase()); // Output: HELLO

//toLowerCase()
String text = "WORLD";
System.out.println(text.toLowerCase()); // Output: world
```

- toUpperCase()
  - Converts letters to uppercase
- toLowerCase()
  - Converts letters to lowercase

```
//trim()
String textWithSpaces = "   Hello   ";
System.out.println(textWithSpaces.trim()); //
Output: Hello
```

- trim()
  - Removes any extra spaces at the beginning or end of a string

```
//replace()
String sentence = "I like cats.";
String newSentence = sentence.replace("cats",
"dogs");
System.out.println(newSentence); // Output: I
like dogs.
```

- replace()
  - Changes all instances of a character or substring to another

Java Packages:

Packages in Java are a mechanism that encapsulates a group of classes, sub-packages and interfaces.

It is said as a namespace that groups the set of related classes and interfaces .

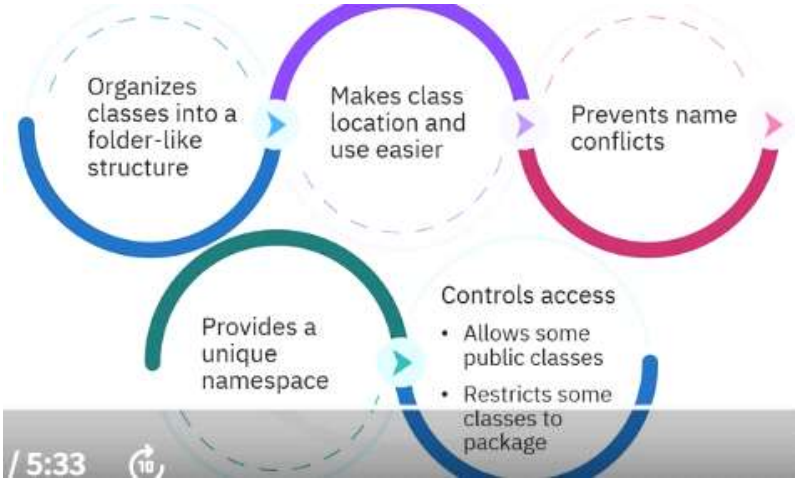
In simple, it works like a folder in a file system.

Note :

namespace refers to a container that holds a set of identifiers (such as variable names, function names, class names, etc.) and allows those identifiers to be organized and managed in a way that avoids naming conflicts.

Key Properties of Packages:

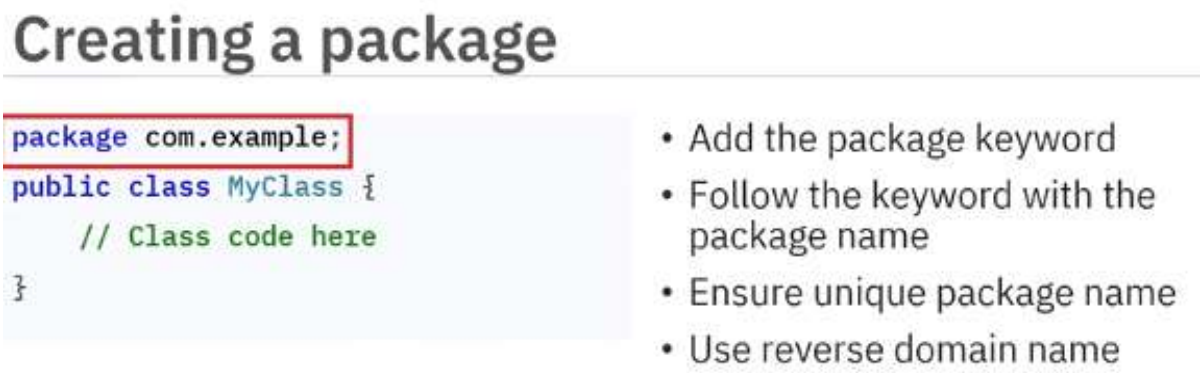
Key Characteristics of Java:



Types of Java Packages:

- User defined packages : These are created by developers to organize their own classes.

How to Declare a Package



Creating a package: Example

```
package shapes;
public class Circle {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    public double area() {
        return Math.PI * radius *
radius;
    }
}
```

- Create a package
- Use package shapes
- Define a class called Circle
- Declare a private radius attribute
- Initialize the radius in the constructor
- Calculate the area of the circle with area() method

Import Statement in Java

Import statement in Java is helpful to take a class or all classes visible for a program specified under a package, with the help of a single statement.

Import: Example

```
import shapes.Circle;
public class TestCircle {
    public static void main(String[]
args) {
        Circle myCircle = new
Circle(5);
        System.out.println("Area of
circle: " + myCircle.area());
    }
}
```

- Use the import statement
- Write the package name
  - shapes.Circle;
- Import the Circle class
  - import shapes.Circle;
- Create an object
  - Circle myCircle = new Circle(5);
- Call methods
  - myCircle.area()

Import all classes

```
import shapes.*; // Import the shapes package
```

- Use an asterisk (*) to import all classes
- Import all classes in shapes package
  - import shapes.*;

- **Built-in-Packages in Java:**

java.lang

- Automatically imported
- Provides core classes

- Key classes
  - String
    - Text manipulation
  - Math
    - Math functions
  - System
    - Input/output and system interaction

This package is automatically imported in every java program.

java.util

- Contains classes for
  - Collections
  - Data structures
  - Algorithms

- Key classes
  - ArrayList
    - Resizable array
  - HashMap
    - Stores key-value pairs
  - Date
    - Manages dates and times

java.io

- Enables input and output operations
- Includes file handling and stream processing

- Key classes
  - File
    - Manages file paths and directories
  - InputStream and OutputStream
    - Reading and writing byte streams
  - BufferedReader
    - Reads text

java.net

- Offers classes for networking
- Allows communication over networks
- Facilitates access to web resources

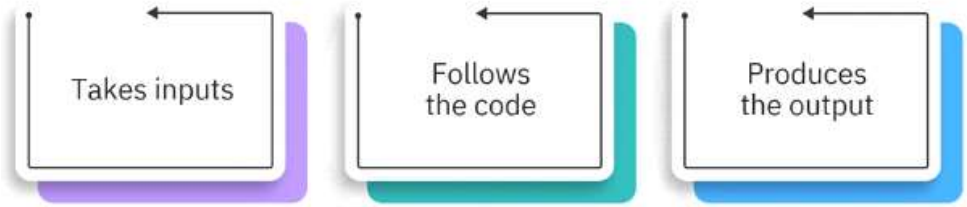
- Key classes
  - Socket
    - Enables client-server communication
  - URL
    - Enables resource access

Functions in Java :

A function is a standalone block of code designed to perform a specific task. It can accept input parameters, process them, and potentially return a value.

Means : Functions are independent of data structures, requiring explicit data passing.

- Functions are the concept of procedural and functional programming languages.
- Functions does not strictly follow the OOP's rules, and are independent not tied to class (or) object.
- And Functions can simply called by there name.



Creating a function

```
returnType functionName(parameter1Type parameter1, parameter2Type parameter2)
{
    // code to be executed

    return value; // optional

}
```

- Return type
  - Specifies the return value type
  - Uses "void" if there is no return value
- Function name
  - Describes the function's task
- Parameters
  - Provide input for data processing
- Code block
  - Contains the logic to execute
  - May include a return statement

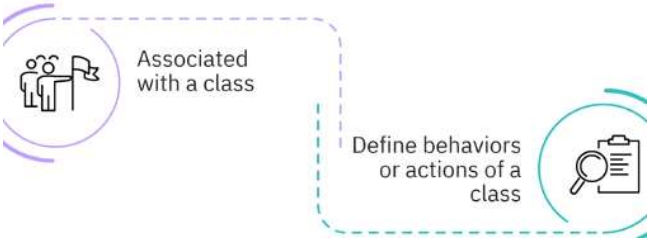
Creating a function: Example

```
public class MathOperations {
    public static int add(int a, int b) {
        return a + b;
    }
    public static void main(String[] args)
{
        int sum = add(5, 3);
        System.out.println("The sum is: "
+ sum);
    }
}
```

- Define the add function with two integers
- Call the add function in the main method
- Store the result in the sum variable
- Print the sum using System.out.println

Method in Java :

- A method is a block of code that belongs to a class or an object.
- It defines the behavior or actions that an object of that class can perform.
- Methods are called using an object instance (e.g., `objectName.methodName()`) or, if they are static, using the class name (e.g., `ClassName.staticMethodName()`).



Creating a method

```
returnType methodName(parameter1Type
parameter1, parameter2Type parameter2) {

    // code to be executed

    return value; // optional

}
```

- Similar to functions
- Includes:
  - Return type
  - Method name
  - Input parameters
  - Code

Creating a method: Example

```
public class Calculator {
    public int multiply(int x, int y) {
        return x * y;
    }
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int product = calc.multiply(4, 5);
        System.out.println("The product is: " +
product);
    }
}
```

- Create a method within a class
- Define the multiply method in the Calculator class
- Instantiate Calculator in the main method to call multiply

Function	Method
Standalone block of code	Tied to a class, operates on its objects
Rarely used in Java	Core to object-oriented programming
Can take inputs and return values	Accesses instance variables and other methods within the class
Not linked to any class or object	Defines behavior and modifies object state

Difference between Parameter and Argument :

Parameter : It is a variable used to define the input type of a Function (or) Method.

Arguments : Are the actual values that are passed to the Function (or) Method when they are called.

Parameters: Example

```
public class Greeting {
    public void greet(String name, int age)
{
        System.out.println("Hello " + name
+ ", you are " + age + " years old.");
    }
    public static void main(String[] args)
{
        Greeting greeting = new Greeting();
        greeting.greet("Alice", 30);
    }
}
```

- Define the greet method:
  - Use String for the name
  - Use int for the age
- Specify the expected values
- Pass the values “Alice” and 30 as arguments in the main method
- Print the personalized greeting:
  - “Hello Alice, you are 30 years old”

Method Overloading :

Method Overloading is a concept that defines multiple methods in the same class but with different parameters (signatures).

This is a form of compile-time polymorphism, where the compiler determines which specific method to invoke based on the method signature (the method's name and its parameter list).

Key characteristics of method overloading:

- **Same Method Name:** All overloaded methods within the same class share the same name.
- **Different Parameter Lists:** The parameter lists of overloaded methods must differ in at least one of the following ways:
  - **Number of parameters:** The methods can have a different count of parameters.
  - **Data types of parameters:** The methods can have parameters of different data types.
  - **Order of parameters:** If the data types are the same, their order can be different.
- **Return Type and Access Modifier:** The return type and access modifier of overloaded methods can be the same or different; they do not play a role in distinguishing overloaded methods.

Method overloading: Example

```
public class Display {
    public void show(int number) {
        System.out.println("Number: " +
number);
    }
    public void show(String text) {
        System.out.println("Text: " + text);
    }
    public static void main(String[] args) {
        Display display = new Display();
        display.show(10);
        display.show("Hello World");
    }
}
```

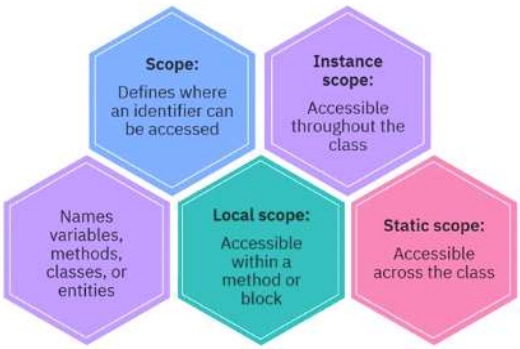
- Define two show methods in the Display class
  - One method takes an int and prints the number
  - The other method takes a String and prints the text
- Create an instance of the Display class
  - Call show with the argument 10
  - Call show with the argument "Hello World"

Identifiers :

In Java, an identifier is a name used to identify and refer to various programming elements within a program. These elements include:

- **Classes:** The name given to a class, e.g., `MyClass`.
- **Methods:** The name given to a method, e.g., `main`, `calculateSum`.
- **Variables:** The name given to a variable to store data, e.g., `age`, `userName`.
- **Interfaces:** The name given to an interface, e.g., `Runnable`.
- **Packages:** The name given to a package that organizes related classes and interfaces.

Identifier



Rules for Java Identifiers:

Starting Character:

An identifier must begin with a letter (A-Z or a-z), a dollar sign (`$`), or an underscore (`_`). It cannot start with a digit.

Subsequent Characters:

After the initial character, an identifier can contain letters, digits (0-9), dollar signs (`$`), or underscores (`_`).

Case Sensitivity:

Java identifiers are case-sensitive. `myVariable` and `MyVariable` are considered different identifiers.

Reserved Keywords:

Identifiers cannot be Java keywords or reserved words (e.g., `public`, `static`, `void`, `class`, `int`, `String`).

Access Modifiers :

Access modifiers in Java are used to control the visibility of the variables, classes, and methods within a class or package.

There are different types of access modifiers that are used to define the accessibility in different ways.

1. public Access Modifier

The public modifier provides the highest access among all the modifiers. We can access it anywhere in the package. The public modifier allows to use the variable, methods and class to acess in the same package.

Example: Demonstration of public access modifier in Java.

```
// Demonstrating the pubic access modifier in Java
import java.io.*;

// public class
public class Geeks
{
    // public method
    public static void main (String[] args) {

        System.out.println("Hello Geeks");
    }
}
```

Example : A **bank's public website** displays general information like **interest rates**, **customer service phone numbers**, and **loan options**. This information is available to everyone.

2. protected Access Modifier

The protected modifier in Java allows the class, method and variable to Accessible within the same package and subclasses.

Example: Demonstrating the **protected** modifier in Java.

```
// Demonstrating the protected modifier in Java
class BaseClass {

    // prptected method
    protected void Msg() {

        System.out.println("This is the protected method.");
    }
}

public class Geeks{

    public static void main(String []args){

      // Creating the object of BaseClass
      BaseClass obj = new BaseClass();
```

Example: An **employee's salary** is private, but a **manager** or **HR staff** (who may extend or be subclasses of the Employee class) should be able to access the salary details for processing.

Or

Consider a **bank manager** who needs to access certain employee details (such as salary) for review. Although salary details are confidential, a manager (who might be a subclass of a BankEmployee class) should have access to it.

3. private Access Modifier

The private access modifier is used with method, class and variable and if the it can be only accessible in the class where it is defined.

Example: Demonstrating the **private** access modifier.

```
// Demonstrating the private modifier in Java
class BaseClass {

    // Private variable
    private String msg = "Hello Geeks";

    // public method having private varialbe
    public void Msg() {

        System.out.println("Private variable having message: "+ msg);
    }
}
```

Example : Real Life :

A customer's **bank account PIN** or **password**. These are extremely sensitive pieces of information that should only be accessible by the owner of the account.

4. default Access Modifier

When the modifier is not define then it automatically specified default method or package private. The default having the accessibility within the same package. If the the variable, method or class having no modifier then it considered as defualt modifier

Example: Demonstrating the default modifier in Java

```
// Demonstrating the default modifier in Java
class BaseClass {

    // default method
     void Msg() {

        System.out.println("This method having defualt modifier.");
    }
}
```

Example: In a bank, **internal transactions** between employees in the same department (like the IT

department) can be handled, but **not by other departments** (like HR or customer service).

Note :

Can you declare a variable without initializing it in java?

Yes, you **can declare a variable without initializing it** in Java, but there are **rules depending on where and how** you declare it.

Difference between Protected and Default access modifier ?

Protected Access Modifier : Where it allows the classes, methods, and variables to be accessible within the package or subclasses of other packages too.

Default Access Modifier : Where it allows the classes, methods, and variables to be accessible within the **own package** only.

Rules : Access Modifiers for Superclass and Subclass:

Superclass Modifier	Subclass Modifier	Explanation
`private`	Cannot be overridden	Private methods and fields are not inherited by the subclass, so they cannot be overridden.
`default`	`default`, `protected`, `public`	The subclass can override the method with `default`, `protected`, or `public`, but not `private`.
`protected`	`protected`, `public`	The subclass can override the method with `protected` or `public` but not `default` or `private`.
`public`	`public`	The subclass can only override with `public` (cannot override with `protected`, `default`, or `private`).

Access Modifiers Summary

Access Modifier	Accessible Within Same Class	Accessible Within Same Package	Accessible in Subclass (Different Package)	Accessible from Anywhere
`private`	Yes	No	No	No
`default`	Yes	Yes	No	No
`protected`	Yes	Yes	Yes (in different package)	No
`public`	Yes	Yes	Yes (in different package)	Yes

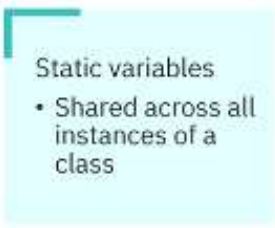
Non Access Modifiers :

In Java, non-access modifiers are keywords that provide additional information about the characteristics and behavior of classes, methods, and variables, without controlling their visibility or accessibility.

1 : static :

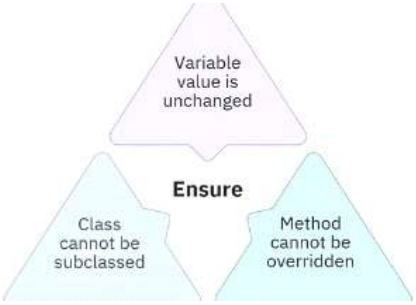
- Indicates that a member's belongs to the class itself rather than to instances of the class.
- Example: If you have a static variable to count the total number of cars created, this variable is shared across all instances of the class.
- Example :
- class Car {
  static int totalCars = 0; // Shared among all Car objects
  }

This count can be shred through the all the objects and can access any static method's or variables without creating an object.



2 : Final Modifier:

- Ensures that a **variable's** value cannot be changed, a **method** cannot be overridden, and a **class** cannot be subclassed.
- Example: If you declare a variable as final, it cannot be modified after its initial




```
public final class Vehicle { // Final
class
    final int maxSpeed = 120; //
Final variable
    final void displayMaxSpeed() { //
Final method
        System.out.println("Max
Speed: " + maxSpeed);
    }
}
```

3 : Abstract Modifier:

- Used for classes and methods.
- **An abstract class** cannot be instantiated, and it serves as a blueprint for other classes.
- **Abstract methods** must be implemented in subclasses.
- Example: An abstract class named Shape can have an abstract method draw() that must be defined in any subclass.

```
public abstract class Shape { // Abstract
class
    abstract void draw(); // Abstract method
}
public class Circle extends Shape {
    @Override
    void draw() { // Implementing abstract
method
        System.out.println("Drawing Circle");
    }
}
}
```

4: synchronized:

- **Methods/Blocks**: Used in multithreading to ensure that only one thread can access a specific method or block of code at a time, preventing data inconsistency.

Note :

To prevent a Java class from being subclassed, Jessica should use the final non-access modifier.

Type of Variable	Can Declare Without Initializing?	Has Default Value?
Instance (non-static field)	✅ Yes	✅ Yes
Static (class field)	✅ Yes	✅ Yes
Local (inside method)	✅ Yes, but must initialize before use	❌ No default

Types of Java Variables

Now let us discuss different types of variables which are listed as follows:

- Local Variables
- Instance Variables
- Static Variables

1. Local Variables

A variable defined within a block or method or constructor is called a local variable.

- The Local variable is created at the time of declaration and destroyed when the function completed its execution.
- The scope of local variables exists only within the block in which they are declared.
- We first need to initialize a local variable before using it within its scope.

2. Instance Variables

Instance variables are known as non-static variables and are declared in a class outside of any method, constructor, or block.

- Instance variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
- Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable. For *String* it is *null*, for *float* it is *0.0f*, for *int* it is *0*, for Wrapper classes like *Integer* it is *null, etc.*
- Scope of instance variables are throughout the class except the static contexts.
- Instance variables can be accessed only by creating objects.
- We initialize instance variables using constructors while creating an object. We can also use instance blocks to initialize the instance variables.

3: Static Variable :

Static variables are also known as class variables.

- These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor, or block.
- Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.
- Initialization of a static variable is not mandatory. Its default value is dependent on the data type of variable. For *String* it is *null*, for *float* it is *0.0f*, for *int* it is *0*, for *Wrapper classes* like *Integer* it is *null*, etc.
- If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won't halt the program. The compiler will replace the object name with the class name automatically.
- If we access a static variable without the class name, the compiler will automatically append the class name. But for accessing the static variable of a different class, we must mention the class name as 2 different classes might have a static variable with the same name.
- Static variables cannot be declared locally inside an instance method.
- Static blocks can be used to initialize static variables.

Generic Programming :

Generic programming is a style of computer programming in which algorithms and data structures are designed to work with various data types without needing to be rewritten for each specific type. This is achieved by using type parameters or placeholders that are specified later when the generic code is used.

Key characteristics of generic programming:

Type Parameterization:

Algorithms and data structures are written using generic type parameters (e.g., `T` in Java or C++, or `KeyType`, `ValueType` in the example below). These parameters act as placeholders for actual types that will be provided when the generic code is instantiated.

Example :

```java
// Generic List class
public class MyGenericList<T> {
    private T[] elements;
    private int size;

    public MyGenericList(int capacity) {
        // ... initialization ...
    }

    public void add(T element) {
        // ... add element ...
    }

    public T get(int index) {
        // ... get element ...
        return null; // Placeholder
    }
}

// Usage with specific types
MyGenericList<Integer> intList = new MyGenericList<>(10);
intList.add(5);

MyGenericList<String> stringList = new MyGenericList<>(10);
stringList.add("Hello");
```

In this example, `MyGenericList<T>` is a generic class where `T` is a type parameter. When `MyGenericList<Integer>` is created, `T` is effectively replaced by `Integer`, and when `MyGenericList<String>` is created, `T` is replaced by `String`. This allows the same `MyGenericList` code to work with different types without modification.

Templets in C++?

In C++, a template is a powerful feature that enables generic programming. It allows you to write functions and classes that can operate on different data types without having to write separate code for each type. Essentially, templates act as blueprints or formulas for creating generic functions or classes.

Example :

Function Templates:

You can define a function template to create a single function that works with different data types. For example, a `max()` function template could find the maximum of two values, regardless of whether they are integers, doubles, or custom objects.

C++

```
template <typename T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}
```

- **Class Templates:** Similarly, you can define a class template to create a class that can hold or operate on different data types. For example, a `Box` class template could store a value of any type.

C++

```
template <typename T>
class Box {
public:
    T value;
    Box(T v) : value(v) {}
    void display() {
        // Code to display the value
    }
};
```

- **Instantiation:** When you use a template (e.g., call a function template or create an object of a class template), the compiler instantiates it by replacing the type parameters with the actual data types you provide. This happens at compile time, leading to type-safe and efficient code generation.

Does Java uses Templates ?

Java, unlike languages such as C++, does not have a feature directly called "templates" in the same compile-time, code-generation sense. However, Java offers several mechanisms and concepts that serve similar purposes or are referred to as "templates" in different contexts:

Generics:

This is Java's closest equivalent to C++ templates for creating reusable code that works with different types. Generics allow you to define classes, interfaces, and methods with type parameters, enabling type-safe code that operates on various data types without needing to write separate code for each type.

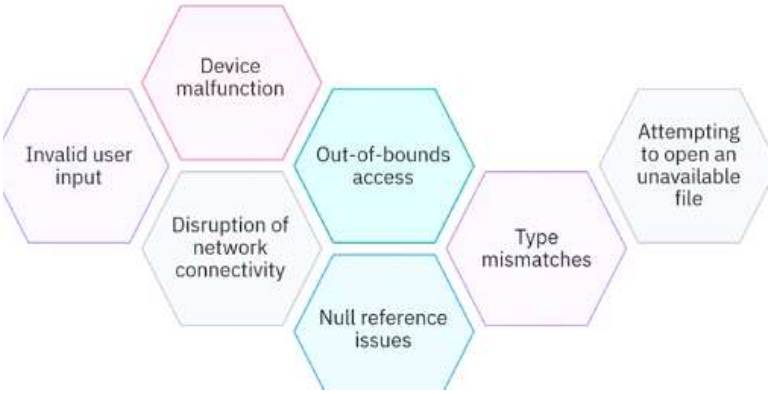
Exceptions in Java :

An exception is defined as an event that stops the normal execution flow of the program.

Exception makes the program to stop / terminate abnormally.

Role of exception is to handle the errors without crashing the program.

Causes of Exceptions :



Types of Exceptions :

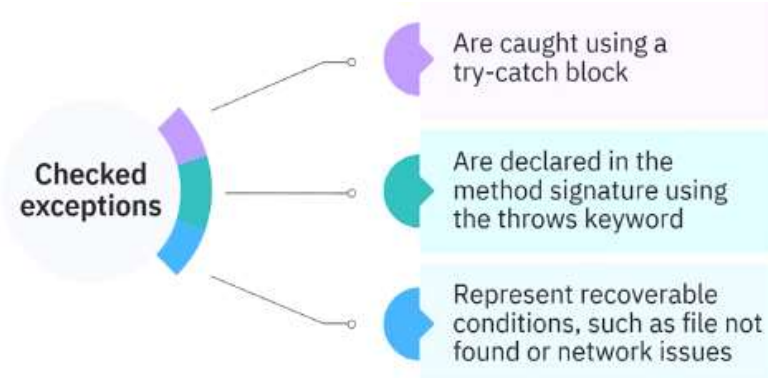
1 : Checked Exception

2 : Un-Checked Exception

1 : Checked Exception:

- These exceptions are checked at compile-time by the compiler, meaning the compiler forces the programmer to handle or declare them.
- These exceptions can be handled by programmer or if the programmer not interested in handling the exception is passed to JVM by using “throw” keyword (which is declared in the method signature using throw keyword).
- Or they can be handled using try-catch block.

Checked exceptions in Java



IOException:
Occurs due to input/output operation failure

SQLException:
Related to database access errors

ClassNotFoundException:
Occurs when an application tries to load a class by its name but cannot find it

◆ **Examples:**

- `IOException`
- `SQLException`
- `FileNotFoundException`
- `ClassNotFoundException`

```
import java.io.*;

public class Demo {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("file.txt"); // Must hand
    }
}
```

How can you handle checked exceptions in your code?

1. Try-Catch Block:

- Wrap the code that may throw a checked exception in a try block.
- Use a catch block to handle the exception if it occurs.

Example :

```
try {
FileReader file = new FileReader("nonexistentfile.txt");
} catch (FileNotFoundException e) {
System.out.println("File not found: " + e.getMessage());
}
```

Throws Keyword:

- Declare the exception in the method signature using the throws keyword. This means that the method can throw the exception, and it must be handled by the calling method.

Example:

```
public void readFile() throws FileNotFoundException {
    FileReader file = new FileReader("nonexistentfile.txt");
}
```

Finally Block:

- You can also use a finally block to execute code that should run regardless of whether an exception occurred or not. This is often used for cleanup activities.

Example:

```
FileReader file = null;
try {
file = new FileReader("nonexistentfile.txt");
} catch (FileNotFoundException e) {
System.out.println("File not found: " + e.getMessage());
} finally {
if (file != null) {
try {
file.close();
} catch (IOException e) {
System.out.println("Error closing the file: " + e.getMessage());
}
}
}
```

2 : Un-Checked Exception :

- These exceptions are raised during the runtime(due to programming error or syntax) and need to be handled at run time only.
- These exceptions are not needed to handled explicitly .

🔷 **Examples:**

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `ArithmeticException`
- `IllegalArgumentException`

🧠 **Example:**

```java
public class Demo {
    public static void main(String[] args) {
        int x = 10 / 0; // ArithmeticException (unchecked)
    }
}
```

NullPointerException:
Occurs when an application attempts to use a null object reference

ArrayIndexOutOfBoundsException:
Thrown when trying to access an index array that's negative or ≥ array size

ArithmeticException:
Occurs when due to an exceptional arithmetic condition

IllegalArgumentException:
Indicates a method has been passed an illegal argument

	Checked Exceptions	Runtime Exceptions
When are they used?	Discovered at compile time	Occur as the program runs
Why identify exceptions?	Handle applications & recover from external events: Files not found, network issues and other issues outside of the code	Identify programming errors for fixes such as log errors, null references, and others
What are the handling requirements?	Must catch to compile the code Use try-catch or declare using throws	No mandatory handling required

	Checked Exceptions	Runtime Exceptions
What are some examples?	IOException, FileNotFoundException, ClassNotFoundException.	NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException
What are some common use cases?	File operations, network communications, database operations and class exceptions	Arithmetic operations, accessing elements in collections, object manipulations

Difference between Error and Exceptions ?

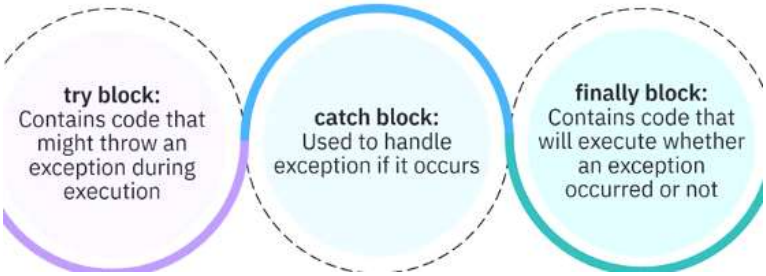
Error : Error are the series of issues that cannot be recovered .

- Types of ErrorsOutOfMemoryError, StackOverflowError
- Errors occurs due to System Faults, etc..

Exception : It is a situation that stops the normal execution flow of program.

- Types of Exceptions : IOException, NullPointerException, ArithmeticException
- Programmer mistakes, incorrect input, Out of Bounds, etc..

Handling Exceptions :



- **try block:**

This block contains the code that might potentially throw an exception.

- **catch block:**

This block immediately follows a try block and specifies the type of exception it can handle. If an exception of that type (or a subclass) is thrown within the try block, the corresponding catch block is executed to handle it.

- **finally block:**

This block is optional and follows a try (and optionally catch) block. The code inside the finally block is always executed, regardless of whether an exception occurred or was caught. It is commonly used for resource cleanup (e.g., closing files or database connections).

- **throw keyword:**

Used to explicitly throw an exception object. This can be used to throw pre-defined exceptions or custom exceptions.

- **throws keyword:**

Used in a method signature to declare that a method might throw one or more specified checked exceptions. This forces the calling method to either handle the declared exceptions or re-declare

them.

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // This will cause an ArithmeticException
            System.out.println("Result: " + result); // This line will not be exe
        } catch (ArithmeticException e) {
            System.err.println("An arithmetic error occurred: " + e.getMessage())
        } finally {
            System.out.println("Finally block executed.");
        }
    }
}
```

try-catch block :

try-catch block is like a safety net for your code.

The try block is where you perform your risky moves, and if you slip (an error occurs), the catch block is there to catch you and prevent you from falling (crashing your program). This way, your program can handle errors gracefully instead of just stopping unexpectedly.

```
try {
  // Code that may throw an exception
} catch (ExceptionType e) {
  // Code to handle the exception
}
```

Java OOPS Concept :

Course objectives

After completing this course, you will be able to:

- Apply Object-Oriented Programming (OOP) concepts
- Define and implement classes and objects, demonstrating encapsulation and abstraction
- Use advanced OOP concepts such as inheritance, polymorphism, interfaces, and method overloading
- Employ the Java collections framework to manage data using lists, sets, queues, and maps
- Read and write to files using Java streams
- Handle directories effectively
- Work with Java date and time classes for effective date manipulation and formatting
- Integrate external services and call external APIs
- Demonstrate your knowledge with the completion of a comprehensive Java programming project

Java Oops :

- **Class**:
    - A class is like a blueprint or template for creating objects.
    - It defines the properties (attributes) and methods (actions) that the objects created from the class will have.
    - Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times.
    - For example, if you have a class called "Dog," it might have properties like "breed" and "color," and methods like "bark" and "fetch."
- **Object**:
    - An object is an instance of a class which defines as is real world entity.
    - It represents a specific item created from the class blueprint.
    - Using the "Dog" class example, an object could be a specific dog named "Buddy" that is a "Golden Retriever" and is "yellow."
    - Each object can have its own unique values for the properties defined in the class.

Object as below features :

- **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by the methods of an object. It also reflects the response of

an object to other objects.

- **Identity**: It is a unique name given to an object that enables it to interact with other objects.

3. Abstraction

- **Abstraction in Java** is the process of hiding the implementation details and only showing the essential details or features to the user.
- It allows to focus on what an object does rather than how it does it.
- The unnecessary details are not displayed to the user.

Note: In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

```
abstract class Vehicle
{
    abstract void speed();
    abstract void breaks();

    void startEngine()
    {
        System.out.println("Engine Started");
    }
}

class Car extends Vehicle
{
    @Override void speed()
    {
        System.out.println("Car is very speed");
    }

    @Override void breaks()
    {
        System.out.println("Car as breaks");
    }
}
public class Main
{
    public static void main(String[] args) {
        Car obj = new Car();
        obj.speed();
        obj.breaks();
        obj.startEngine();
    }
```

4. Encapsulation

- Encapsulation is defined as the process of wrapping data and the methods into a single unit, typically a class.
- It is the mechanism that binds together the code and the data.
- Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Encapsulation
  - Protects sensitive information
  - Defines private properties
  - Provides controlled access

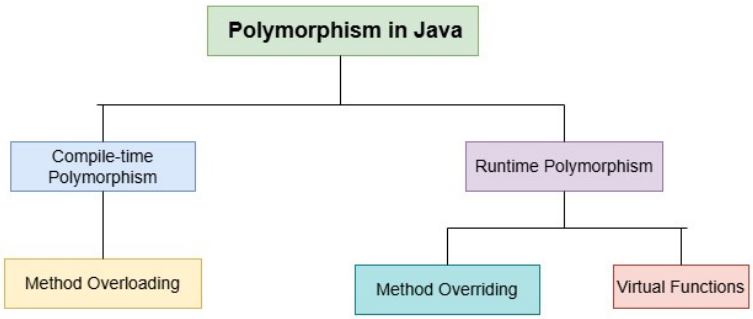
```java
class Employee
{
    private int id;
    private String name;

    void data(int id, String name)
    {
        this.id = id;
        this.name = name;
    }

    void display()
    {
        System.out.println("Employee name is "+name + " and id is "+id);
    }
}

public class Main
{
    public static void main(String[] args) {
        Employee obj = new Employee();
        obj.data(10,"Abc");
        obj.display();
    }
}
```

5. Inheritance

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using **extends** keyword. Inheritance is also known as "**is-a**" relationship.

6. Polymorphism

The word polymorphism means having **many forms**, and it comes from the Greek words **poly (many)** and **morph (forms)**, this means one entity can take many forms. In Java, polymorphism allows the same method or object to behave differently based on the context, specially on the project's actual runtime class.



Constructor:

A Constructor is a special method that help to create and set up object.

Set up means : constructor organizes and initializes the different attributes of an object when it is created.

Need of Constructors :

1 : For proper initialization of object

2 : Flexible object creation with different states like (providing values in object passed to parameterized contractor).

Constructor Chaining:

Constructor Chaining in Java

Constructor Chaining in Java is the process of calling one constructor from another within the same class or from a parent class.

There are **two types** of constructor chaining:

1. **Within the same class** (using `this(...)`)
2. **From a subclass to a superclass** (using `super(...)`)

Example :

1 : Within the Same Class using this keyword

```java
class Bank
{
    String name;
    int balance;

    public Bank()
    {
        this("Unknown", 0);
        System.out.println("This is Default Constructor");
    }

    public Bank(String name, int balance)
    {
        this.name = name;
        this.balance = balance;
        System.out.println("This is Parameterized Constructor");
    }
    void display() {
        System.out.println("Name: " + name + ", Balance: " + balance);
    }
}
```

```java
public class Main
{
    public static void main(String[] args) {
        Bank obj = new Bank();
        obj.display();

        Bank obj2 = new Bank("abc",20);
        obj2.display();
    }
}
```

```
This is Parameterized Constructor
This is Default Constructor
Name: Unknown, Balance: 0
This is Parameterized Constructor
Name: abc, Balance: 20
```

Execution Steps :

1 : 1st with object reference “obj” default constructor is called, in default constrctor with the help of “this” keywordparameterized constructor is called with 2 values.

2 : 2nd step as the parameter constructer is called the parameter constructor is executed and the values are assigned to the variables of “obj” reference and go back to default constructor then default constructor is executed.

And obj.display() method is called and the values are displayed.

3: 3rd step directly the parameterized constructor is called with “obj2” refand it is executed.

Then obj2.display() method is executed.

2 : From the subclass to superclass using super keyword.

```java
class Account {
    String name;

    Account(String name) {
        this.name = name;
        System.out.println("Account constructor");
    }
}

class BankAccount extends Account {
    double balance;

    BankAccount(String name, double balance) {
        super(name); // calls parent constructor
        this.balance = balance;
        System.out.println("BankAccount constructor");
    }

    void display() {
        System.out.println("Name: " + name + ", Balance: " + balance);
    }
}
```

```java
public class Main
{
    public static void main(String[] args) {
        BankAccount obj = new BankAccount("abc", 20);
        obj.display();
    }
}
```

```
Account constructor
BankAccount constructor
Name: abc, Balance: 20.0
```

Copy Constructor in Java

A **copy constructor** is a special constructor used to **create a new object by copying the fields of another object** of the same class.

Purpose of a Copy Constructor:

To create a **new object** with the **same values** as an existing object.

Syntax (User-defined)

java

```
ClassName(ClassName otherObject) {
    this.field1 = otherObject.field1;
    this.field2 = otherObject.field2;
    // copy all relevant fields
}
```

Example :

```
class BankAccount {
    String name;
    double balance;

    // Parameterized constructor
    BankAccount(String name, double balance) {
        this.name = name;
        this.balance = balance;
    }

    // 🔁 Copy constructor
    BankAccount(BankAccount other) {
        this.name = other.name;
        this.balance = other.balance;
    }
```

```
    void display() {
        System.out.println("Name: " + name + ", Balance: " + balance);
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount original = new BankAccount("Alice", 1000);
        BankAccount copy = new BankAccount(original); // using copy constructor

        original.display();  // Name: Alice, Balance: 1000.0
        copy.display();      // Name: Alice, Balance: 1000.0
    }
}
```

Q1 : Can a Java constructor be private?

Yes, a Java constructor can be declared as `private`.

A private constructor restricts the instantiation of a class from outside the class itself. This means that you cannot directly create objects of a class with a private constructor using the `new` keyword from other classes.

Buffer Overflow :

Buffer is a portion of RAM which is fixed amount of memory (temporary memory) used to store the data.

If the program tries to store more data than the buffer size (capacity) can accommodate, the excess data can overwrite adjacent memory locations.

This is called buffer overflow.

Means : Buffer size is usually fixed for a given program. But what if a program tries to write more data than the allocated memory? This usually results in Buffer Overflow.

Simply put, **Buffer Overflow refers to a program writing content to memory outside of the buffer** (as a result of overflow).

Note :

Does C+ provide a default copy constructor ?

Yes, C++ provides a default copy constructor for a class if no user-defined copy constructor is explicitly provided. This default copy constructor performs a member-wise copy of the data members from the source object to the newly created object.

Implicitly generated:

The compiler automatically generates this constructor when a copy constructor is not explicitly defined by the programmer.

Member-wise copy:

It copies the value of each non-static data member from the source object to the corresponding member in the destination object.

Constructor Practice :

- **Module 1**: Create the basic classes for employees and leave requests
- **Module 2**: Expand the system with inheritance and polymorphism for different types of leave

Employee Class :

```
class Employee
{
    String name;
    int id;
    int leaveBalance;

    public Employee(String name, int id, int leaveBalance)
    {
        this.name = name;
        this.id = id;
        this.leaveBalance = leaveBalance;
    }

    void display()
    {
        System.out.println("Name :"+name+" id :"+id+" leaveBalance :"+leaveBalance);
    }

    String getName()
    {
        return name;
    }

    int getId()
    {
        return id;
    }
```

```
    void detectLeave(int days)
    {
        if(days <= leaveBalance)
        {
            leaveBalance = leaveBalance - days;
        }
        else
        {
            System.out.println("Insufficent leaves");
        }
    }

    void addLeaves(int days)
    {
        leaveBalance = leaveBalance + days;
    }

    public int getLeaveBalance()
    {
        return leaveBalance;
    }
}
```

Leave Request class :

```java
class LeaveRequest
{
    int requestId;
    Employee emplyee;
    int NoOFDays;
    String status;

    public LeaveRequest(int requestId, Employee emplyee, int NoOFDays, String status)
    {
        this.requestId = requestId;
        this.emplyee = emplyee;
        this.NoOFDays = NoOFDays;
        this.status = "Pending";
    }

    int getRequestId() {return requestId;}
    public Employee getemployee() { return emplyee;}
    int getDays() { return NoOFDays;}
    String getStatus() { return status ;}

    void show()
    {
        System.out.println("Request ID :"+requestId +" Employee"+emplyee.getName()+" NoOFDays :"+NoOFDays+
        " status :"+status);
    }
```

```java
    void approved()
    {
        if(emplyee.getLeaveBalance() >= NoOFDays)
        {
            emplyee.detectLeave(NoOFDays);
            status = "approved";
        }
        else{
            status = "rejected ( no balance )";
        }
    }
    void reject()
    {
        status = "reject";
    }
}

public class Main
{
    public static void main(String[] args) {
        Employee emp = new Employee("abc", 10, 1000);
        LeaveRequest leaveRequest =  new LeaveRequest(123, emp, 5, "pending");
        leaveRequest.show();
        // Step 4: Approve the leave
        leaveRequest.approved();
        leaveRequest.show();
        // Step 5: Check employee's leave balance
        emp.display();
    }
}
```

Output :

```
Request ID :123 Employeeabc NoOFDays :5 status :Pending
Request ID :123 Employeeabc NoOFDays :5 status :approved
Name :abc id :10 leaveBalance :995
```

Practice :

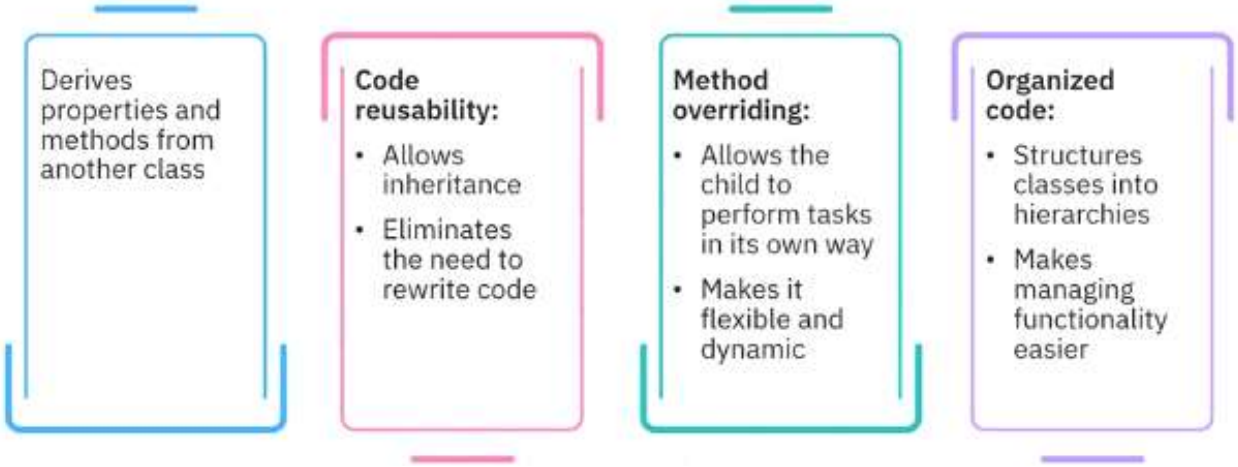
Your learning journey

As you progress through each module, you'll gradually build different components of the Leave Tracking System:

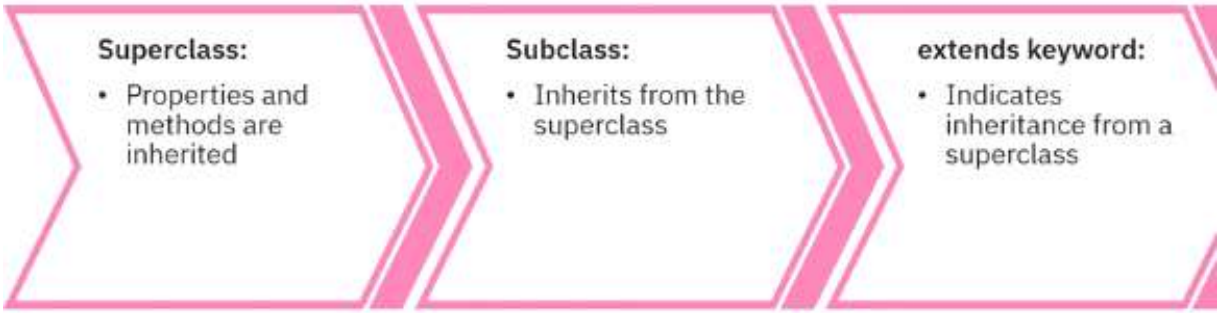
- **Module 1**: Create the basic classes for employees and leave requests
- **Module 2**: Expand the system with inheritance and polymorphism for different types of leave
- **Module 3**: Implement Java collections to store and manage multiple requests
- **Module 4**: Use file operations to save and load leave data
- **Module 5**: Implement date handling for leave periods and calculations
- **Module 6**: Complete the leave system with the creation of a functional application

Inheritance :

That allows a new class (subclass or child class) to derive properties and behaviors (fields and methods) from an existing class (superclass or parent class).



Inheritance Terminologies :



Types of Inheritance Supported in Java :

1: Singel Inheritance

2 : Multilevel Inheritance

3 : Hierarchal Inheritance

Interview Questions on Inheritance :

1: What does Java's inheritance mean?

Inheritance is one of the piller of OOPS, That allows a new class (subclass or child class) to derive properties and behaviors (fields and methods) from an existing class (superclass or parent class).

- **Inheritance reduces the code reusability :** No need to rewrite the same code again and again in multiple classes, which reduces the code efficiency and redundancy .
- **Method Overriding (Runtime Polymorphism):**

Inheritance is fundamental to achieving method overriding, which enables runtime polymorphism. Method overriding allows a subclass to provide a specific implementation for a method that is already defined in its superclass.

How does Java implement or achieve inheritance?

Two keywords can be used to implement or accomplish inheritance:

- **extends:** The keyword extends is used to create an inheritance relationship between two classes and two interfaces.
- **implements:** The term "implements" establishes the line of descent between a class and an interface.

What is the use of extends and implements keyword in java ?

In Java, extends and implements are keywords used to establish relationships between classes and interfaces, enabling inheritance and polymorphism.

1. extends Keyword:

The extends keyword is used to create a subclass from an existing class, establishing an inheritance relationship.

It signifies an "is-a" relationship, meaning the subclass "is a type of" the superclass.

- Extends keyword also overcome the “diamond problem” and maintain in the inheritance hierarchy .

```
class Animal {
    void eat() {
        System.out.println("Animal eats.");
    }
}

class Dog extends Animal { // Dog extends Animal
    void bark() {
        System.out.println("Dog barks.");
    }
}
```

2. implements Keyword:

The implements keyword is used by a class to implement one or more interfaces. It signifies that the class agrees to provide implementations for all the abstract methods declared in the interface(s).

```
interface Playable {
    void play();
}

class MusicPlayer implements Playable { // MusicPlayer implements Playable
    @Override
    public void play() {
        System.out.println("Music is playing.");
    }
}
```

S.No.	Extends	Implements
1.	By using "extends" keyword a class can inherit another class, or an interface can inherit from other interfaces	By using "implements" keyword a class can implement an interface
2.	It is not compulsory for a subclass that extends a superclass to override all the methods in a superclass.	It is compulsory for a class implementing an interface to implement all the methods of that interface.
3.	A class can extend only one super class	A class can implement any number of an interface at the same time
4.	Any number of interfaces can be extended by interface.	An interface can extend other interfaces but cannot implements them.

What is Diamond Problem in JAVA ?

The diamond problem in Java refers to an ambiguity issue that can arise in object-oriented programming when a class attempts to inherit from multiple parent classes that share a common ancestor, and each of these parent classes potentially provides a different implementation of a method inherited from the common ancestor. This creates a "diamond" shape in the inheritance hierarchy.

Line-by-Line Explanation:

✓ Line 1:

The diamond problem in Java refers to an ambiguity issue that can arise in object-oriented programming

- **What this means:**
  The *diamond problem* is a classic issue in **multiple inheritance**, where **ambiguity** arises. Java is an object-oriented language, and this problem shows up **when you try to inherit from more than one class**.

✓ Line 2:

...when a class attempts to inherit from multiple parent classes

- **What this means:**
  Suppose you create a class (C) that tries to extend **two different classes** (A and B).

Example:

- class A { void show() {...} }
- class B extends A { void show() {...} }
- class C extends A, B { ... } // ✕ Not allowed in Java

Java does **not allow** this kind of multiple class inheritance because of ambiguity.

✓ Line 3:

...that share a common ancestor

- **What this means:**
  Both parent classes (e.g., B and C) are themselves subclasses of a **common base class** (A).
  So, there's a shared ancestor in the hierarchy.

✓ Line 4:

...and each of these parent classes potentially provides a different implementation of a method inherited from the common ancestor.

- **What this means:**
  If both B and C override a method (like show()) from the base class A,
  and another class D inherits both B and C, then **which version of show() should it inherit?**
  This creates a **conflict**, or **ambiguity**.

✓ Line 5:

This creates a "diamond" shape in the inheritance hierarchy.

- **What this means:**
  Here's a visual:
- A
- / \
- B C
- \ /
- D
- The class D inherits from both B and C, and both inherit from A.
  This structure looks like a **diamond**, hence the name **"diamond problem"**.

❗ Why This Is a Problem:

- If B and C both override A's method, then class D will be **confused**:
    - Should it use B's version of the method?
    - Or C's?

Collections :

In Java, Collections refer to a framework that provides a **set of classes and interfaces** to store and manipulate a group of data elements.

It is part of the java.util package and provides various data structures like: lists, sets, maps, and queues, along with utility methods to operate on them.

Let's break down the difference between:

✅ 1. Collections API

✅ 2. Collection (interface)

✅ 3. Collections (class)

✅ 1. Collections API

- **Definition**: The **Collections API** refers to the **entire framework** of interfaces, classes, and algorithms provided in the `java.util` package to handle **groups of objects**.
- **It includes**:
  - Interfaces: `Collection`, `List`, `Set`, `Queue`, `Map`, etc.
  - Implementations: `ArrayList`, `HashSet`, `HashMap`, `TreeSet`, etc.
  - Algorithms: Sorting, searching, shuffling, etc. (from the `Collections` class).
- **Goal**: To provide ready-to-use data structures and utility methods for working with groups of objects.

> 📌 Think of **Collections API** as the **whole toolbox** of data structures and utilities in Java.

Interfaces :

Collection : The root interface in the collection hierarchy, it represents a group of objects.

List: An ordered collection that can contain duplicates. Examples include ArrayList, LinkedList.

Set: A collection that does not allow duplicate elements. Examples include HashSet, LinkedHashSet.

Queue: A collection used to hold elements before processing. Examples include PriorityQueue, LinkedList (implements Queue).

Map: An object that maps keys to values, where each key is associated with exactly one value. Examples include HashMap, TreeMap.

Implementations :

These are the actual classes that implement the above interfaces and provide concrete

functionality. For example:

ArrayList: A resizable array that implements the List interface.

HashSet: A set that uses a hash table for storage, offering fast lookups.

LinkedList: A doubly-linked list implementation of both the List and Queue interfaces.

HashMap: A map that stores key-value pairs using a hash table.

✅ 2. Collection (Interface)

- **Definition:** `Collection` is the **root interface** of the **collection hierarchy** (except for maps). It defines basic methods like:
  - `add()`, `remove()`, `size()`, `clear()`, `contains()`, `iterator()`, etc.
- **Extended by:**
  - `List`
  - `Set`
  - `Queue`

> 📌 It **does not include** `Map`, because maps store key-value pairs, not just elements.

Example:

```java
Collection<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
System.out.println(names);
```

✅ 3. Collections (Class)

- **Definition**: `Collections` (note the **s**) is a **utility class** that provides **static methods** to operate on or return collections.
- It's part of the `java.util` package.
- Common methods:
  - `Collections.sort(List)`
  - `Collections.reverse(List)`
  - `Collections.shuffle(List)`
  - `Collections.max(Collection)`
  - `Collections.unmodifiableList(List)` (to make it read-only)

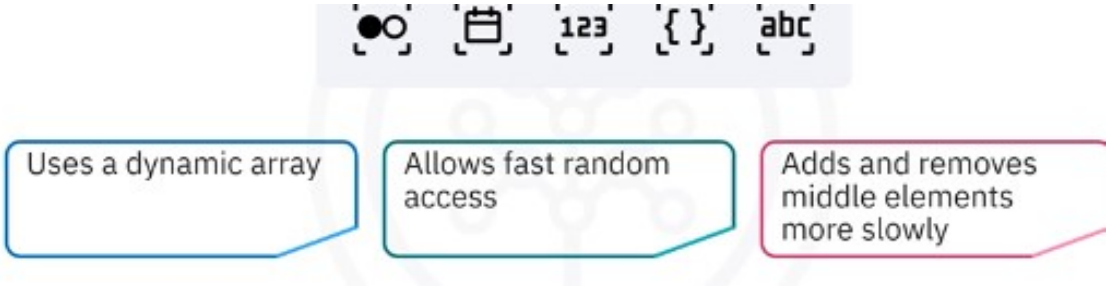
Example :

```java
import java.util.*;

public class CollectionsDemo {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Banana");
        fruits.add("Apple");
        fruits.add("Mango");

        Collections.sort(fruits);  // Sort list
        System.out.println("Sorted: " + fruits);

        Collections.reverse(fruits);  // Reverse list
        System.out.println("Reversed: " + fruits);
    }
}
```

Types of Java Collections

Types	Collection type	Capabilities	Common implementations
Lists	Ordered	• Allow duplicate elements • Maintain order	• ArrayList • LinkedList
Sets	Unordered	• Do not allow duplicate elements	• HashSet • TreeSet
Maps	Unordered	• Key-value pairs where each key is unique	• HashMap • TreeMap

1: ArrayList :



ArrrayList is a part of List collection framework.

Think of it like a dynamic array — unlike regular arrays in Java (String[] arr = new String[10];), an ArrayList can grow and shrink automatically as elements are added or removed.

🔷 Key Points About ArrayList

Feature	Description
Type	Class (implements `List`, which extends `Collection`)
Package	`java.util`
Ordering	Maintains insertion order
Duplicates	Allows duplicate elements
Null values	Allowed
Resizable	Yes — it grows/shrinks automatically
Random access	Fast — uses an internal array

```java
import java.util.ArrayList;
import java.util.*;

public class ArrayLists {
    Run | Debug
    public static void main(String[] args) {
        ArrayList<Integer> nums = new ArrayList<>();
        nums.add(e:10);
        nums.add(e:20);
        nums.add(e:30);

        System.out.println("Nums :" + nums);
        int value = nums.get(index:1);
        System.out.println(value);
        nums.set(index:2, element:50);
        System.out.println(nums);
        // nums.remove(0);
        nums.size();
        nums.isEmpty();
        System.out.println(nums.contains(o:10));

        // iterate, for each and Iterator , use any one
        for(Integer i : nums)
        {
            System.out.println(i*2);
        }
        Iterator<Integer> iterator = nums.iterator();
        while(iterator.hasNext())
        {
            System.out.println(iterator.next());
        }
    }
```

In Java, the `ArrayList` class (part of the `java.util` package) provides many **useful operations**. Here's a categorized list of the **common operations** you can perform on an `ArrayList`, along with examples:

🔷 **1. Create an ArrayList**

```java
ArrayList<String> list = new ArrayList<>();
```

📋 Java ArrayList Cheatsheet

🔹 Feature	💻 Code Example
1. Create an ArrayList	`ArrayList<String> list = new ArrayList<>();`
2. Add Elements (end)	`list.add("Apple");`
2. Add Elements (index)	`list.add(1, "Banana");`
3. Access Elements	`String fruit = list.get(0);`
4. Modify Elements	`list.set(0, "Mango");`
5. Remove by Index	`list.remove(1);`
5. Remove by Object	`list.remove("Apple");`
6. Size of the List	`int size = list.size();`

7. Check if Empty	`boolean isEmpty = list.isEmpty();`
8. Contains (Search)	`boolean hasApple = list.contains("Apple");`
8. Index Of	`int index = list.indexOf("Banana");`
9. Clear the List	`list.clear();`
10. Iterate (for-each)	`for (String fruit : list) { System.out.println(fruit); }`
10. Iterate (for loop)	`for (int i = 0; i < list.size(); i++) { System.out.println(list.get(i)); }`
10. Iterate (iterator)	`Iterator<String> it = list.iterator(); while (it.hasNext()) { System.out.println(it.next()); }`
11. Convert to Array	`String[] array = list.toArray(new String[0]);`
12. Sort List	`Collections.sort(list);`
13. Reverse List	`Collections.reverse(list);`
14. Copy List	`ArrayList<String> newList = new ArrayList<>(list);`

🔷 Internal Working (How it Works Under the Hood)

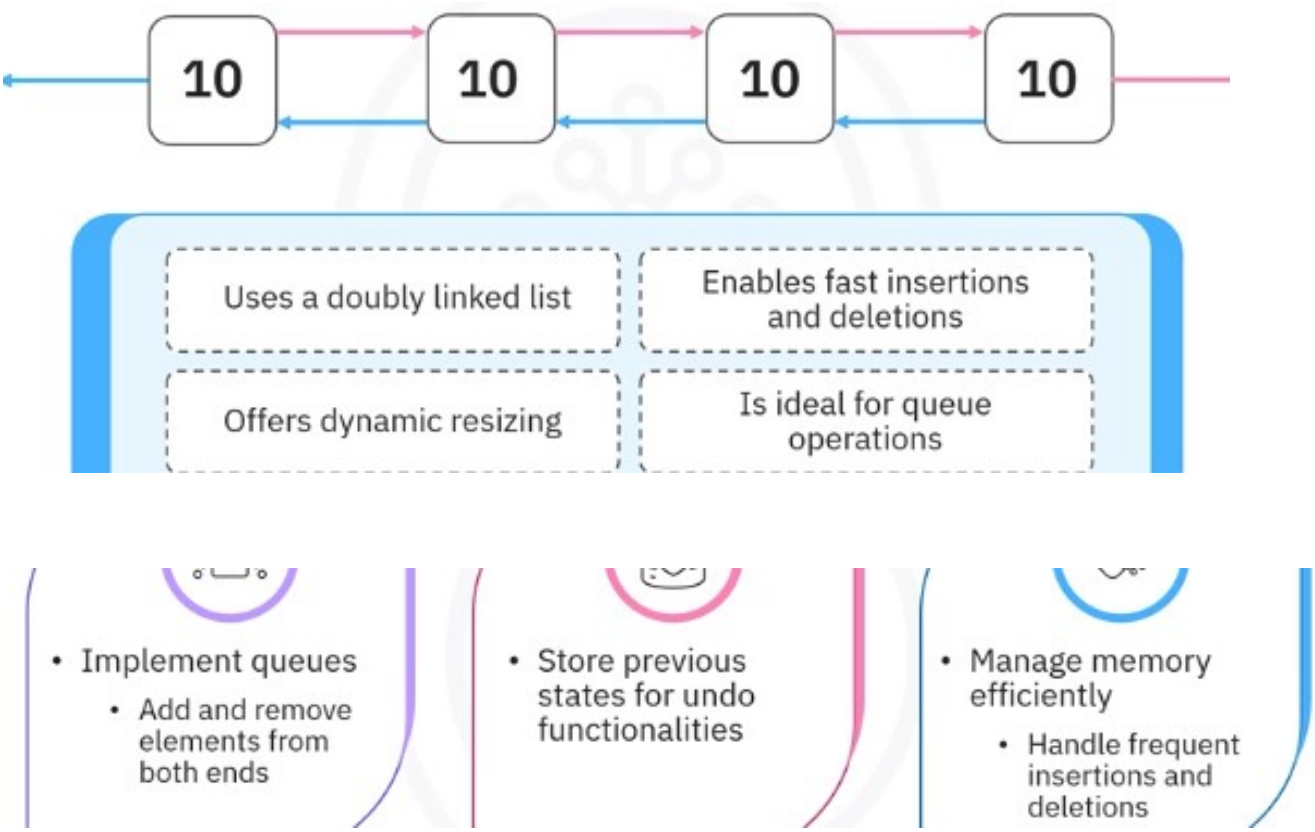
- Internally, `ArrayList` uses a **regular array**.
- When the array becomes full, it **creates a new, larger array**, and **copies the old elements** into it.
- By default, the initial capacity is **10**, and it grows by **1.5x** when needed.

✅ Example:

```java
ArrayList<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Cherry");

System.out.println("Fruits: " + fruits); // [Apple, Banana, Cherry]
System.out.println("Size: " + fruits.size());
System.out.println("Contains Banana? " + fruits.contains("Banana"));
fruits.remove("Apple");
System.out.println("After removal: " + fruits);
```

Let me know if you want a cheat sheet or operations for `ArrayList` of custom objects or numbers.

LinkedList:



A LinkedList in Java is a doubly-linked list implementation of the List and Deque interfaces. It is part of the Java Collections Framework, and defined in the java.util package.

Unlike an ArrayList, which uses a dynamic array internally, a LinkedList consists of a chain of nodes, where each node contains:

- The data (element).
- A reference to the previous node.
- A reference to the next node.

v chat

Key Features of LinkedList

Feature	Description
Type	Class (implements `List`, `Deque`, `Queue`)
Package	`java.util`
Ordering	Maintains insertion order
Duplicates	Allowed
Null values	Allowed
Resizable	Yes
Random access	❌ Slow — unlike `ArrayList`
Insert/Delete	✅ Fast at beginning or middle
Thread-safe?	❌ No — not synchronized by default

v chat

Common Methods of LinkedList

Method	Description
`add(E e)`	Adds to end of list
`addFirst(E e)`	Adds to the beginning
`addLast(E e)`	Adds to the end
`remove()` / `removeFirst()`	Removes first element
`removeLast()`	Removes last element
`getFirst()` / `getLast()`	Gets first/last element
`get(int index)`	Returns element at index
`size()`	Returns number of elements
`clear()`	Removes all elements

```java
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.add("Horse");

        System.out.println("Animals: " + animals);
    }
}
```

🔷 Internal Working (How it Works)

- Each node has:
  - A **value**.
  - A **reference to previous node**.
  - A **reference to next node**.

> This makes adding/removing nodes **faster at the start or middle**, but **slower for accessing elements by index**, because you must traverse the list node-by-node.

🔷 When to Use LinkedList?

Use `LinkedList` when:

- You need **frequent insertions/deletions** at the beginning or middle of the list.
- You don't need **fast random access**.

Feature	ArrayList	LinkedList
Storage	Uses a dynamic array	Uses doubly linked nodes
Access speed	Faster	Slower (requires traversal)
Modification speed	Slower (shifting/resizing)	Faster (no shifting needed)
Initial capacity	Default capacity of 10	Starts empty
Best use case	Minimal modifications	Frequent modifications

HashSet

What is a Set in Java?

A Set is a collection that:

- Does NOT allow duplicate elements
- Does not guarantee order (depends on implementation)
- Allows only one NULL value
- Is backend by HashMap
- Is part of the Java Collections Framework
- Is an interface, not a class

Key Features of Set:

Feature	Description
Duplicates	❌ Not allowed
Null values	✅ Allowed (only one) in most implementations
Ordering	❌ Not guaranteed (unless using `LinkedHashSet` or `TreeSet`)
Interface	Yes – it's the base interface for `HashSet`, `TreeSet`, etc.

Common Set Implementations:

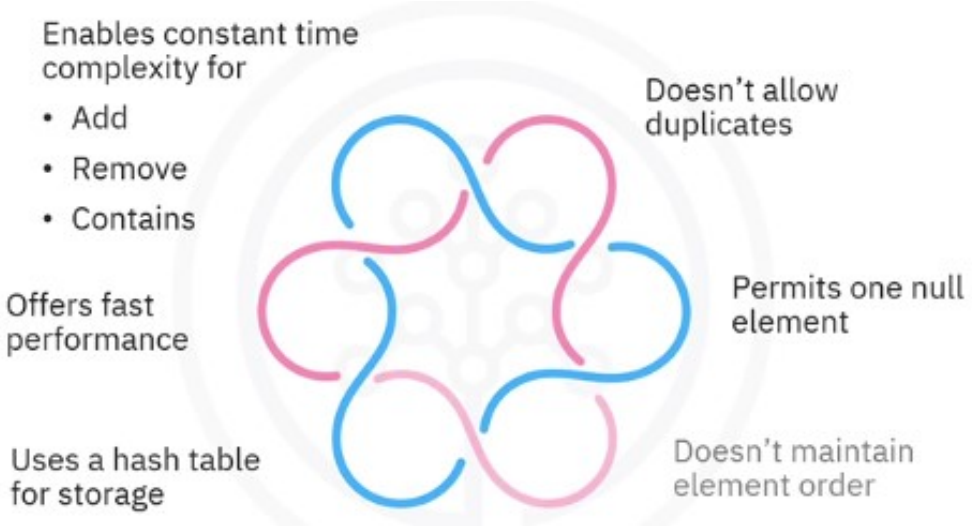
- `HashSet` – No order
- `LinkedHashSet` – Maintains insertion order
- `TreeSet` – Sorted order

What is HashSet in Java?

A HashSet is the most commonly used implementation of the Set interface. It uses a hash table for storage.

Feature	Description
Implements	`Set` interface
Duplicates	❌ Not allowed
Ordering	❌ Unordered (no guarantee of insertion order)
Null values	✅ Allows one `null`
Underlying structure	Uses a **HashMap** internally
Performance	Fast for add, remove, and lookup (O(1) average)

As Hashset does not maintain any order, so accessing of a particular element with index poition from the hashset is difficult.



Example :

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

public class HashSetExample {
    public static void main(String[] args) {
        Set<String> names = new HashSet<>();

        names.add("Alice");
        names.add("Bob");
        names.add("Alice");  // Duplicate, will be ignored
        names.add("Charlie");

        System.out.println("Names: " + names);
    }
}
```

✅ When to Use HashSet?

Use `HashSet` when:

- You want to store **unique items**
- You don't care about **order**
- You need **fast operations** (like add, remove, contains)

Feature	Behavior
Duplicates	❌ Not allowed
Null values	✅ Allowed (only one)
Insertion order	❌ Not preserved
Index-based access	❌ Not supported (use `List` for that)
Thread-safe	❌ No (use `Collections.synchronizedSet()` if needed)
Underlying structure	Uses **hashing** via `HashMap`

--

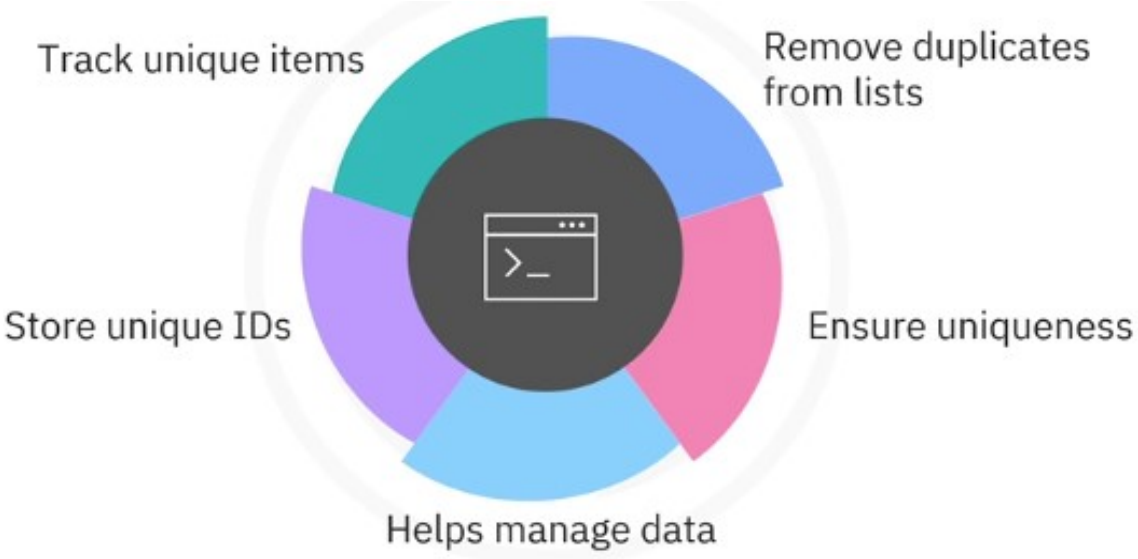
🔧 Common Operations on `HashSet`

Here's a list of commonly used operations:

Operation	Method	Example
--- --	---	---
Add element	`add(E e)`	`set.add("Apple")`
Remove element	`remove(Object o)`	`set.remove("Apple")`
Check existence	`contains(Object o)`	`set.contains("Apple")`
Get size	`size()`	`set.size()`

Check if empty	`isEmpty()`	`set.isEmpty()`
Remove all	`clear()`	`set.clear()`
Iterate	`for` / `for-each` / iterator	`for (String s : set)`
Union	`addAll(Collection)`	`set1.addAll(set2)`
Intersection	`retainAll(Collection)`	`set1.retainAll(set2)`
Difference	`removeAll(Collection)`	`set1.removeAll(set2)`

HashSet Applications :



🆚 `HashSet` vs `ArrayList` vs `TreeSet`

Feature	`HashSet`	`ArrayList`	`TreeSet`
Duplicates	❌ No	✅ Yes	❌ No
Order maintained	❌ No	✅ Yes (insertion)	✅ Sorted
Null elements	✅ One allowed	✅ Multiple allowed	✅ One al
Access by index	❌ Not possible	✅ Yes	❌ Not po

🔥 Real-World Use Cases

* Remove duplicates from a list
* Fast lookups where order doesn't matter
* Store a set of unique IDs, tags, or values

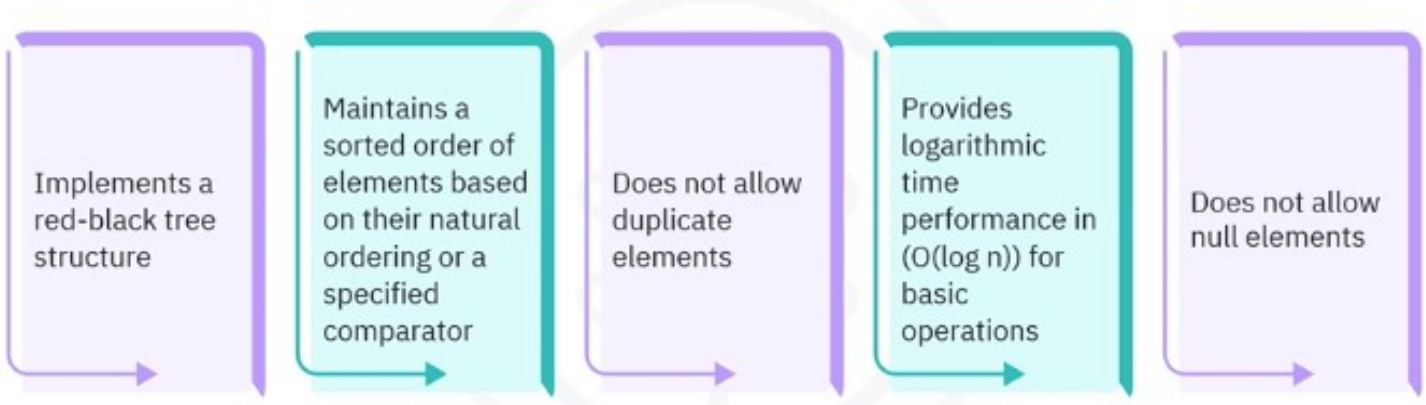
TreeSet :

✅ What is TreeSet in Java?

A **TreeSet** is a class that implements the `Set` interface and stores **unique elements** in **sorted (asce order.**

➤ Key Features of TreeSet:

Feature	Description
Implements	`NavigableSet` , `SortedSet` , `Set`
Ordering	✅ Sorted in **natural order** (or custom using `Comparator`)
Duplicates allowed?	❌ No – duplicates are ignored
Null values?	❌ Not allowed (will throw `NullPointerException`)
Thread-safe?	❌ No
Underlying structure	**Red-Black Tree** (self-balancing binary tree)
Performance	O(log n) for `add` , `remove` , `contains`



🔷 Example: TreeSet

```java
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();

        numbers.add(30);
        numbers.add(10);
        numbers.add(20);
        numbers.add(10);  // Duplicate, ignored

        System.out.println("Sorted Set: " + numbers);
    }
}
```

✅ When to Use TreeSet?

Use `TreeSet` when:

- You want to **store unique elements**
- You need elements to be **automatically sorted**
- You don't need to allow `null` values

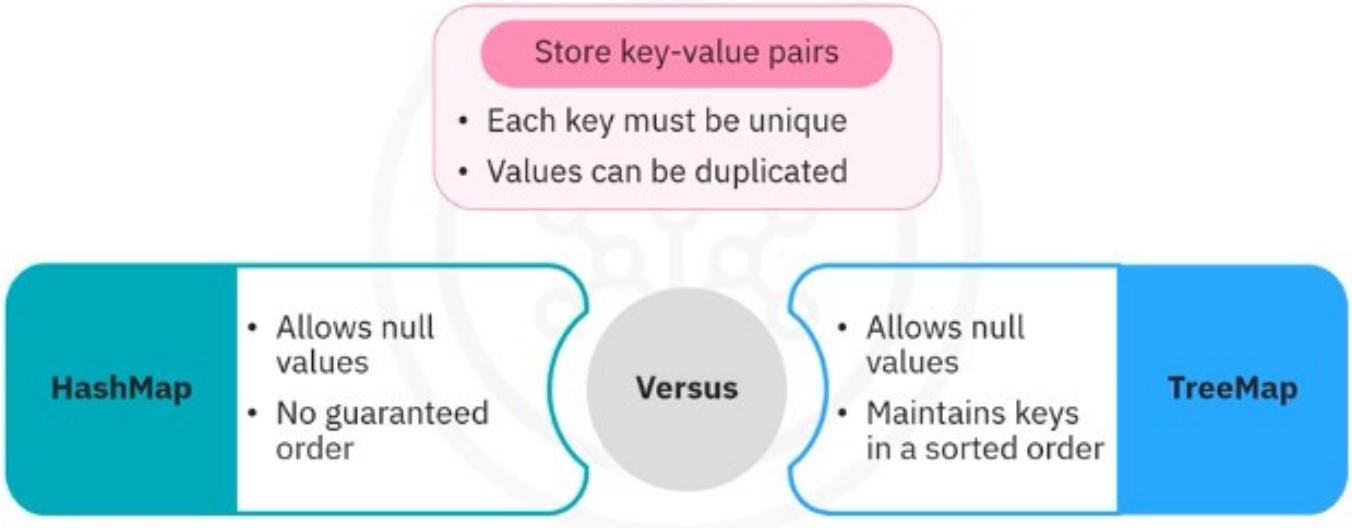
HashSet versus TreeSet

Feature	TreeSet	HashSet
Order of elements	Elements are sorted	No specific order
Speed	Slower (operations take longer)	Fast (quick operations)
Duplicates	Does not allow duplicates	Does not allow duplicates
Null values	Cannot store null values	Can store one null value
Data structure	Uses a balanced tree	Uses a hash table

Map :

a

Using Map collections HashMaps



HashMaps versus TreeMaps

Feature	HashMap	TreeMap
Ordering	Unordered	Ordered (sorted by keys)
Null handling	Allows one null key and multiple null values	Does not allow null keys
Data structure	Uses a hash table	Uses a red-black tree
Memory usage	Generally uses less memory	Generally uses more memory
Iteration order	No guaranteed order	Iterates in key order

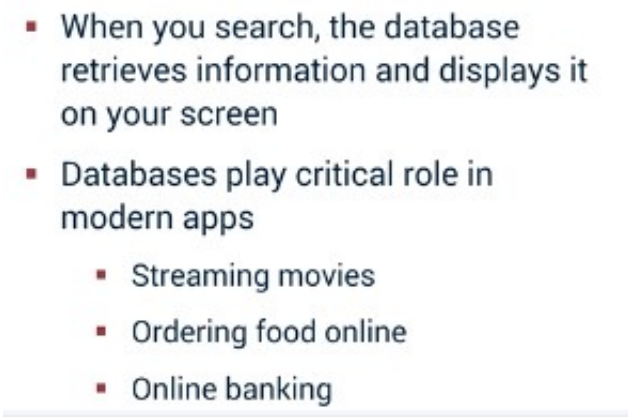
DataBase:

A Database is a structred way to store, manage, and retrieve data. And the data is tored and managed Electronically.

- Organized collection of data stored and managed electronically
- Data is structured to make it easy to work with

  Example - Amazon
- Every product listing, customer account, order, and delivery status is stored in a database

, and when you search for any product (or) item.



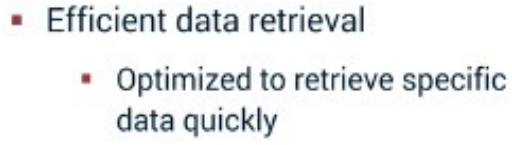
app --> applications

The need Databases:

While storing data in a simple file might work for small applications, but it become inefficient and provides errors as the data grows (or for using larger applications).

DataBase solves this problem by offering the following features :

1 : Efficient Data Retrieval : Can easily retrieve the specific data from the database.



2 : Data Integrity : Ensure Data remains accurate and consistent(even when multiple users access it simultaneously).

3: Scalability : Databases can handle large volumes of data as the applications grow.

4: Security : Database is designed with built in features to control access and protect sensitive information.

Example :

Instagaram Application :

which need,

- Needs to store billions of pieces of data:
  - User profiles
  - Posts
  - Comments
  - Likes
  - Messages
- Database ensures data is stored in organized manner
- App can display your feed quickly and accurately

The Main ability of Databases should be :

1 : Handle Transaction : Databases must be able to handle transactions, which is a series of operations executed as a single unit.
Example :

Online Banking -- > Money Transfering from one account to another account.

And if any erros occurs, then the databases should rolback the changes to maintain consistency.

2: Concurrent Access : Meaning multiple users can read and write data at the same time. This is essential for applications like ticket booking systems, where thousands of people miht try to nook seats simultaneously.

Types of Databases :

1 : Relational Databases :

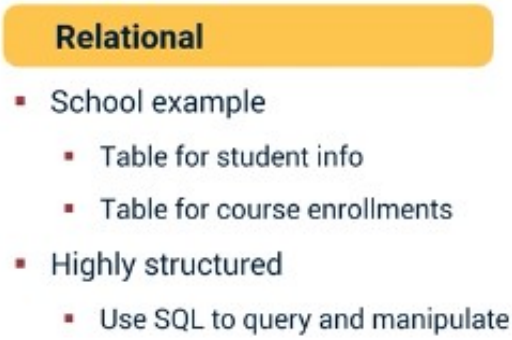
Where data is storesd in tables which includes rows, columns, and relationship between tables which helps to organize the complex datasets .

Relational Databases stores only the structred data (data which is associated with rows and columns (like in a tables)).

Language: SQL

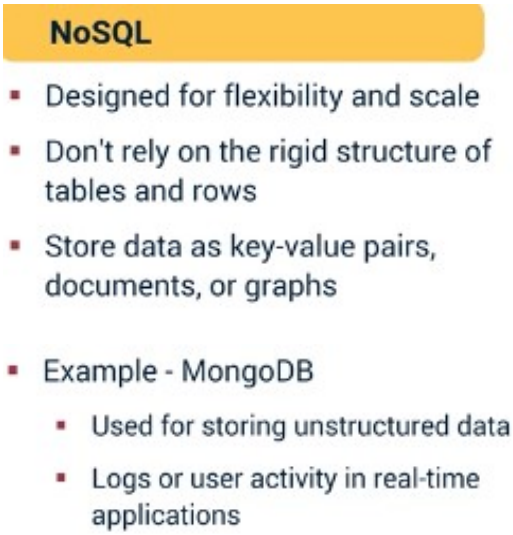
Example :

In a School enrollment.



2: NoSQL Databases :

Which stores un-structured data (data is not stored in tables), but in Key-Value pairs.



3: Graph Databases

4: Time Series Database :

Time series

- Used to store and analyze time-stamped data
- Server logs, stock market prices, or IoT sensor data
- Allows developers to track trends
- Example - Monitor energy usage in smart home

Selecting the right database

- Depends on the application you're building
- Inventory management system
  - Relational database
  - MySQL
- Weather forecasting app
  - Time series database
  - InfluxDB

RDBMS fundamentals

- Database system that organizes data into tables
- Rows and columns
  - Rows represent individual data records
  - Columns represent fields/attributes

StudentID	Name	Age	Major
101	Alice Carter	22	Computer Science
102	John Smith	21	Mathematics
103	Maria Lopez	23	Physics

Primary keys

- Set of columns in a table that uniquely identify each record
- Example: 'StudentID' column can serve as primary key
- Ensure there are no duplicate records in a table

Foreign keys

- A column that references primary key in another table
- Creates a relationship between tables
- Example: Enrollments table

EnrollmentID	StudentID	CourseName
1	101	Data Structures
2	102	Linear Algebra
3	101	Operating Systems

- 'StudentID' acts as foreign key
- Links each enrollment to a student in 'Students' table
- Relationship shows which students are on which course

Indexes

- Database feature that speeds up data retrieval
- Create index for students by StudentID
  - With no index, database would need to scan every row
- Especially helpful with large databases
- Trade-off is slowing down of insert and update operations

Relationships in RDBMS

- One-to-one
  - 'Person' table with a 'Passport' table
- One-to-many
  - 'Customer' table with 'Orders' table
- Many-to-many
  - 'Students' table with 'Courses' table
  - Usually implemented with join table (for example, 'Enrollments')

SQL :

The basics of SQL

- Structured Query Language (SQL) is standard for communications with relational databases
- SQL can create tables, insert data, retrieve data
- We'll focus on DDL, DML, and Procedures

What is DDL?

- Set of commands to define database structure
- How we create, modify, and delete database objects
- Most common DDL commands:
  - CREATE – create new objects
  - ALTER – modify existing objects
  - DROP – delete objects

DDL example

- Setting up database for online bookstore
- To create a table for book information:

```
CREATE TABLE Books (
BookID INT PRIMARY KEY,
Title VARCHAR(100) NOT NULL,
Author VARCHAR(100),
Price DECIMAL(10, 2)
);
```

- Creates 'Books' table with 4 columns: BookID, Title, Author, Price
- PRIMARY KEY ensures each book has unique ID
- NOT NULL means title cannot be empty

DDL example (continued)

- Add new column for publication year
- Use ALTER command:

```
ALTER TABLE Books

ADD PublicationYear INT;
```

- Use ADD with column name and data type
- To remove table with DROP command:

```
DROP TABLE Books;
```

✅ Common Operations Using `ALTER TABLE`

Here's a table of the most commonly used operations:

Operation	SQL Syntax Example
1. Add a new column	`ALTER TABLE table_name ADD column_name data_type;`
	➤ `ALTER TABLE students ADD email VARCHAR(100);`
2. Drop (delete) a column	`ALTER TABLE table_name DROP COLUMN column_name;`
	➤ `ALTER TABLE students DROP COLUMN age;`
3. Rename a column	`ALTER TABLE table_name RENAME COLUMN old_name TO ne`
	➤ `ALTER TABLE students RENAME COLUMN full_name TO`

4. Change column data type

`ALTER TABLE table_name MODIFY COLUMN column_name new_data_type;` *(MySQL)*

➤ `ALTER TABLE students MODIFY COLUMN name VARCHAR(150`

(PostgreSQL uses `ALTER COLUMN ... TYPE` *instead)*

➤ `ALTER TABLE students ALTER COLUMN name TYPE VARCHAR`

5. Rename the table

`ALTER TABLE old_table_name RENAME TO new_table_name;`

➤ `ALTER TABLE students RENAME TO learners;`

6. Add a constraint

`ALTER TABLE table_name ADD CONSTRAINT constraint_name constraint_definition;`

7. Drop a constraint

`ALTER TABLE table_name DROP CONSTRAINT constraint_name;` *(PostgreSQL/Oracle)*

➤ `ALTER TABLE students DROP CONSTRAINT pk_id;`

8. Set a default value

`ALTER TABLE table_name ALTER COLUMN column_name SET DEFA default_value;`

➤ `ALTER TABLE students ALTER COLUMN age SET DEFAULT 18;` *(PostgreSQL)*

9. Drop default value

`ALTER TABLE table_name ALTER COLUMN column_name DROP DEF`

➤ `ALTER TABLE students ALTER COLUMN age DROP DEFAULT;`

What is DML?

- Set of commands to interact with data records in a table
- DDL sets up data; DML manages data
- Primary DML commands:
  - INSERT – adds new records
  - UPDATE – modifies existing records
  - DELETE – removes records
  - SELECT – retrieves records
    - Technically part of Data Query Language (DQL)

DML example

- Add a new book with the INSERT command:

```
INSERT INTO Books (BookID, Title, Author, Price)

VALUES (1, 'The Art of SQL', 'Stephane Faroult', 39.99);
```

- Adds new book to 'Books' table
- If the price changes, use UPDATE command:

```
UPDATE Books

SET Price = 29.99

WHERE BookID = 1;
```

- WHERE clause ensures only BookID 1 is updated

DML example (continued)

- Remove a book with the DELETE command and the WHERE clause:

```
DELETE FROM Books

WHERE BookID = 1;
```

- Retrieve all books written by an author
- Use SELECT command with WHERE clause as a filter:

```
SELECT * FROM Books

WHERE Author = 'Stephane Faroult';
```

✅ Common Operations with `UPDATE`

Here's a table of common things you can do:

Operation	SQL Example
1. Update a single column	`UPDATE employees SET salary = 50000 WHERE id = 1;`
2. Update multiple columns	`UPDATE employees SET salary = 60000, position = 'Manager' W = 2;`
3. Update all rows	`UPDATE employees SET active = true;` *(Updates every row! Be careful)*

4. Update with a condition	`UPDATE students SET grade = 'A' WHERE score >= 90;`
5. Update using values from another column	`UPDATE products SET price = price * 1.10;` *(Increase price b*
6. Update with subquery	`UPDATE employees SET department_id = (SELECT id FROM depa WHERE name = 'HR') WHERE id = 5;`
7. Update using JOIN (advanced)	`sql UPDATE employees e SET e.department = d.name FROM dep d WHERE e.dept_id = d.id;` *(PostgreSQL syntax)*

Note :

We can use UPDATE Command by combining Logical Operators like : (AND, OR, NOT)

✅ Using `UPDATE` with Logical Operators (`AND`, `OR`, `NOT`)

These operators help you apply the `UPDATE` to **specific rows** based on **multiple conditions**.

Logical Operator	Description	Syntax Example	Result
`AND`	All conditions must be **true**	`UPDATE employees SET salary = 60000 WHERE dept = 'IT' AND experience > 5;`	Updates salary only for IT emp with more than 5 years of exp
`OR`	At least one condition must be **true**	`UPDATE employees SET bonus = 1000 WHERE dept = 'Sales' OR performance = 'Excellent';`	Updates bonus for Sales emp those with Excellent performa
`NOT`	Reverses the condition (negates it)	`UPDATE users SET active = false WHERE NOT verified;`	Deactivates users who are no
Combination	Combine `AND`, `OR`, `NOT` with brackets `()`	`UPDATE orders SET status = 'Delayed' WHERE (delivery_days > 7 AND priority = 'Low') OR NOT paid;` ↓	Updates orders based on con condition

Aggregation Functions;
Funtions

Clauses :

GroupBy

Joins :

What are Procedures?

- Stored procedure is a reusable set of SQL statements
- Encapsulate logic to repeatedly perform complex operations
- Similar to functions in programming
  - Takes input, performs operation, returns a result

Procedures example

- Calculate total price of all books in 'Books' table
- Use the CREATE PROCEDURE command:

```
CREATE PROCEDURE CalculateTotalPrice()
BEGIN
 SELECT SUM(Price) AS TotalPrice
 FROM Books;
END;
```

- Call this procedure whenever needed with CALL command:

```
CALL CalculateTotalPrice();
```

Common scenarios for procedures

Use procedures where you need to:

- Automate repetitive tasks
  - Monthly reports
- Perform complex business logic
  - Calculating discounts
  - Validating data
- Enhance performance
  - Reduce data sent between apps and database

Example : Procedures, in Banking System,

- Procedures for banking:
  - Calculate monthly interest on all accounts
  - Take inputs such as account type and interest rate
  - Perform calculations and update account balances

Note :

2 or more procedures creation at a time are not allowed in MYSQL , and also 2 or more procedures calls at a time is not allowed in mysql.

JDBC

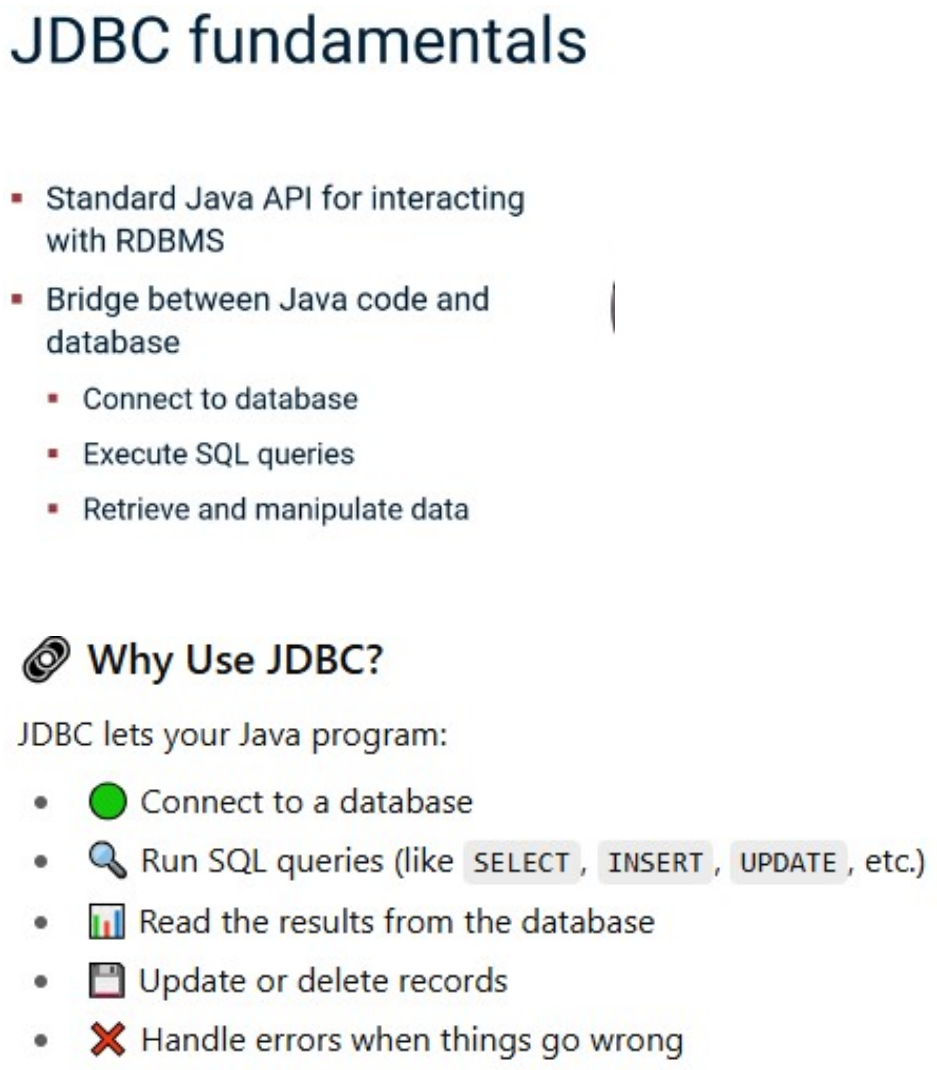
What you will learn

- Understand what Java Database Connectivity (JDBC) is
- Describe the JDBC architecture and its key components
- Establish a database connection using JDBC
- Describe the importance of connection pooling
- Implement a basic connection pool using a library

JDBC stands for Java Database Connectivity.

It is a Java API (Application Programming Interface) that allows Java programs to connect to and

interact with databases.



Understand with an example: why do we need JDBC ?

Imagine you're **building an e-commerce website**, where users can browse :

- products,
- place orders, and
- track deliveries

To make this happen, java application needs to store and retrieve data, such as : user details (or) product details from the database.

JDBC provides a standardized way to perform these operations.

Diagram :

JDBC Architecture Components:

1: Driver Manager : Driver Manages is a **class** that manages a list of Database Drivers.

It establishes connection between the java application and the database.

Role:

- It **loads** and **registers** JDBC drivers.
- It provides a method called `getConnection()` to establish a **connection to a database** using the appropriate driver.

Example:

java

```
Class.forName("com.mysql.cj.jdbc.Driver"); // Load MySQL driver
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/school", "root", "password");
```

2: Driver : A driver is a specific implementation for a particular database. Example, MySQL Driver, PostgreSQL Driver.

◆ Role:

- Converts the **JDBC API calls** from your program into **database-specific calls**.
- Every database (MySQL, Oracle, etc.) provides its **own driver**.

◆ Examples of Drivers:

Database	Driver Class Name
MySQL	`com.mysql.cj.jdbc.Driver`
PostgreSQL	`org.postgresql.Driver`
Oracle	`oracle.jdbc.driver.OracleDriver`
SQL Server	`com.microsoft.sqlserver.jdbc.SQLServerDriver`

3: Connection : A connection represents the connection to the database. It's the starting point for executing SQL queries.

◆ Role:

- Once the driver connects to the database, it returns a `Connection` object.
- It's used to **create statements**, **start transactions**, and **close connections**.

◆ Example:

java Copy

```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/school", "root", "password");
```

4: Statements : Statements and PreparedStatements are objects used to execute SQL queries.

Statements : are simple sql queries.

Prepared Statements : are parameterized queries .

◆ Example (Statement):

java Copy code

```java
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM students");
```

◆ Example (PreparedStatement):

java Copy code

```java
PreparedStatement ps = con.prepareStatement("SELECT * FROM students WHERE id =
ps.setInt(1, 10);
ResultSet rs = ps.executeQuery();
```

5: ResultSet : Stores the data retrieved from the database.

◆ Role:

- Stores **records (rows)** fetched from the database.
- You can **loop through** the ResultSet to read data from each row.

◆ Example:

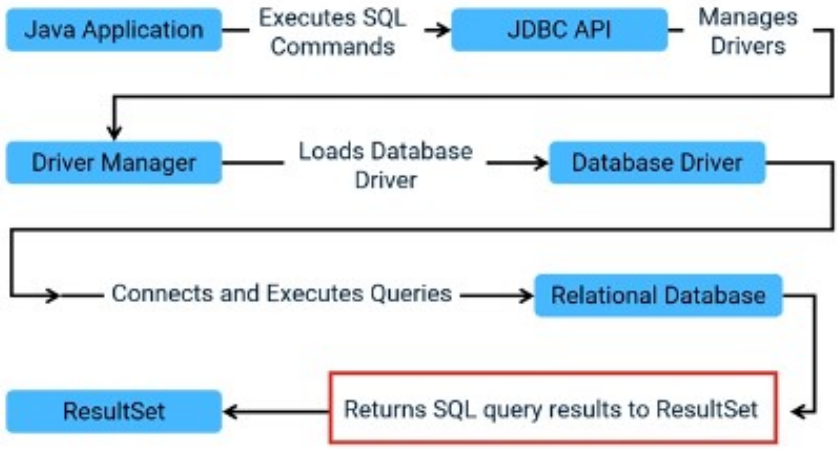
java

```java
while (rs.next()) {
    System.out.println("Name: " + rs.getString("name"));
    System.out.println("Age: " + rs.getInt("age"));
}
```

- DriverManager
  - Establishes connection between app and database
- Driver
  - Specific implementation of a database (MySQL, PostgreSQL)
- Connection
  - Starting point for executing SQL queries
- Statements / PreparedStatements
  - Objects used to execute SQL queries
- ResultSet
  - Stores data retrieved from database

Visula Flow of JDBC Architecture Components Flow :

- Java app communicates with DriverManager
- DriverManager uses appropriate driver to connect to database
- Statements used to execute SQL queries
- Data retrieved by queries is stored in ResultSet



Steps :

Step 1 : First, your Java application communicates with DriverManager through the JDBC API, then the

Step 2: DriverManager component selects the appropriate driver to connect to the database (loading the appropriate drivers).

Step 3: Once connected, a statement is used to execute SQL queries on the relational database.

Step 4 : And finally, the data retrieved by queries is stored in a result set for processing.

✅ JDBC Component Flow Diagram

```
pgsql

1. Java Program
       ↓
2. DriverManager (selects appropriate driver)
       ↓
3. Driver (talks to DB)
       ↓
4. Connection (opens link to DB)
       ↓
5. Statement / PreparedStatement (executes SQL)
       ↓
6. ResultSet (holds query results)
```

How JDBC Works :

1 : Import the packages

2: Load the drivers

Load the JDBC Driver

```
java

Class.forName("com.mysql.cj.jdbc.Driver");
```

3: Register the Drivers (optional)

4: Create a Connection :

```
java

Connection con = DriverManager.getConnection(...);
```

5: Create a Statement :

```java
Statement stmt = con.createStatement();
```

6: Execute the Query using Statement:

```java
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

7: Process the result :

```java
while (rs.next()) {
    System.out.println(rs.getString("name"));
}
```

8: Close the resources :

```java
rs.close();
stmt.close();
con.close();
```

Connection Pool :

Without a connection pool, each user request would open and close a new database connection, leading to slow response times and resource exhaustion.

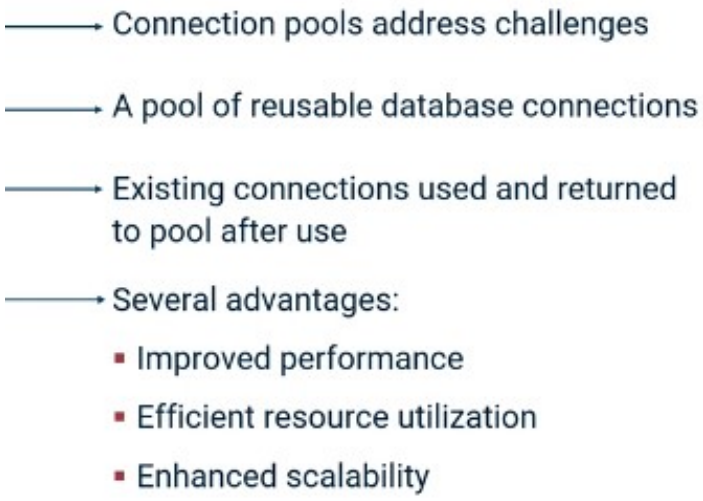
If each user request initiates a new database connection, the system faces several challenges.

1. First would be **high latency**: Establishing a new database connection for every request

introduces significant overhead, leading to slower response times.

2. Next would be **resource exhaustion:** Databases have a finite capacity for concurrent connections. Rapidly opening and closing connections can quickly exhaust available resources, causing failures or degraded performance.
3. And lastly, there would be **scalability issues:** As the user base grows, the system may struggle to handle the increased load, resulting in poor user experience and potential revenue loss.

Defination :



By using a connection pool, you can handle high traffic efficiently with minimal database overhead. Connection pooling addresses the challenges discussed by maintaining a pool of reusable database connections.

When a request is made, an existing connection is assigned from the pool, and upon completion, it's returned for future use.

This approach offers several advantages:

1: Improved performance:

Reusing established connections eliminates the overhead of creating and closing connections, resulting in faster data retrieval and transaction execution.

2: Efficient resource utilization:

Pooling allows for more efficient use of database resources, reducing the likelihood of resource

exhaustion, and ensuring stable performance under heavy load.

3: Enhanced scalability:

With a managed pool of connections, the application can handle a higher number of simultaneous user requests, supporting business growth without compromising performance.

Steps to Establish a Connection :

Step1 : Add Database Driver : For MySQL we sue = MySQl Connector / J

For Maven we write the dependency code.

- Add the database driver
  - MySQL = **MySQL Connector/J**
  - Maven = Add code below to **pom.xml**

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-
java</artifactId>
    <version>8.0.33</version>
</dependency>
```

Step2 : Write the Connection Code:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.*;
public class DatabaseConnection
{
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/bookstore";
        String user = "root";
        String password = "password";

        try {
            Connection conn = DriverManager.getConnection(url, user, password);
            if(conn != null)
                System.out.println(x:"Connected to the database!");
        } catch (SQLException e) {
            // TODO: handle exception
            System.out.println(x:"Error connecting to the database");
            e.printStackTrace();
        }
    }
}
```

How to implement Connnection pooling using Apache Commons DBCP :

Apache Commons DBCP is an Apache Commons project library for managing a pool of reusable database connections.

Step 1 : First, you add this dependency code section to your pom.xml file.

```xml
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.9.0</version>
</dependency>
```

Step 2 : Java Code to implement connection pool:

```
public class ConnectionPoolExample {
    public static void main(String[] args) {
        // Create a connection pool
        BasicDataSource dataSource = new BasicDataSource();

dataSource.setUrl("jdbc:mysql://localhost:3306/bookstore");
        dataSource.setUsername("root");
        dataSource.setPassword("password");
        dataSource.setInitialSize(5); // Number of initial
connections in the pool
```

```
        try (Connection connection =
dataSource.getConnection();
             Statement statement =
connection.createStatement();
             ResultSet resultSet =
statement.executeQuery("SELECT * FROM Books")) {
```

while(resultSet.hasnext())

{

S.O.P(resultSet.next());

}}

JDBC Statements :

Introduction to JDBC Statements

- Enable you to execute SQL queries and interact with databases
- Basic queries to advanced concepts
  - Filtering
  - Sorting
  - Joining tables
  - Handling NULLs
- Code examples and explanations

JDBC Statements enable you to execute SQL queries (static sql queires [without parameters]) and interact with databases.

Queries that can be executed using Statements are : SELECT, INSERT, UPDATE, AND DELETE.

We can also perform operations like :

Filtering

Sorting

Joining tables and

Handling NULL values .

Type	Class	Use Case
1 Statement	`java.sql.Statement`	For static SQL (hardcoded queries)

◆ Syntax:

java Copy code

```java
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM students");
```

◆ Example:

java Copy code

```java
Statement stmt = con.createStatement();
int result = stmt.executeUpdate("UPDATE students SET grade='A' WHERE id=1");
```

Example : Code .

- Code example: Retrieve all rows from 'Books' table
- Three data retrieval methods:
  - getInt
  - getString
  - getDouble

```java
ResultSet resultSet =
statement.executeQuery("SELECT * FROM Books");

while (resultSet.next()) {

  System.out.println("Book ID: " +
resultSet.getInt("BookID"));

  System.out.println("Title: " +
resultSet.getString("Title"));

  System.out.println("Author: " +
resultSet.getString("Author"));

  System.out.println("Price: " +
resultSet getDouble("Price"));

}
```

1: Filtering data with WHERE clause:

Filtering data with a WHERE clause

- WHERE filters rows based on specified conditions
- Code example: Retrieves books by specific author
- WHERE clause is essential for retrieving targeted data

```
ResultSet resultSet = statement.executeQuery(
  "SELECT * FROM Books WHERE Author = 'Author X'");
while (resultSet.next()) {
  System.out.println("Title: " +
resultSet.getString("Title"));
}
```

2: Sorting results using ORDER BY:

- ORDER BY sorts query results based on columns
- Code example: Retrieves books sorted by descending price
- If you omit DESC, default order is ascending (ASC)

```
ResultSet resultSet = statement.executeQuery(
  "SELECT * FROM Books ORDER BY Price DESC");
while (resultSet.next()) {
  System.out.println("Title: " +
resultSet.getString("Title"));
  System.out.println("Price: " +
resultSet.getDouble("Price"));
}
```

Note :

Can also sort by multiple columns :

```
SELECT * FROM Books ORDER BY Price DESC,
Title ASC;
```

where you can see Price is Sorted by DESC, Title is Sorted by ASC.

Note :

We can use LIMIT to restrict the number of records :

Limiting results with the LIMIT clause

- LIMIT restricts number of result rows returned
- Useful for result previewing and pagination
- Code example: Retrieves top 5 most expensive books
- ORDER BY is a common technique for 'Top N' lists

```
ResultSet resultSet = statement.executeQuery(
  "SELECT * FROM Books ORDER BY Price DESC
LIMIT 5");
while (resultSet.next()) {
  System.out.println("Title: " +
resultSet.getString("Title"));
  System.out.println("Price: " +
resultSet.getDouble("Price"));
```

3: Aggregate data with GROUP BY :

- GROUP BY to group rows with same value
- Apply aggregate functions (COUNT, SUM, AVG)
- Code example: Counts number of books by each author
- Use GROUP BY when creating summary reports
  - Total sales per region
  - Number of products sold by category

```
ResultSet resultSet = statement.executeQuery(
  "SELECT Author, COUNT(*) AS BookCount FROM
Books GROUP BY Author");
while (resultSet.next()) {
  System.out.println("Author: " +
resultSet.getString("Author"));
  System.out.println("Number of Books: " +
resultSet.getInt("BookCount"));
}
```

4: Joining tables with JOIN :

- JOIN combines rows from two or more tables
- Code example: Retrieves book titles and their publishers
- Use JOIN to combine data from related tables
  - Orders with customer details
  - Products with their suppliers

```
ResultSet resultSet = statement.executeQuery(
  "SELECT Books.Title, Publishers.Name " +
  "FROM Books " +
  "INNER JOIN Publishers ON
 Books.PublisherID = Publishers.PublisherID");
while (resultSet.next()) {
  System.out.println("Book: " +
resultSet.getString("Title"));
  System.out.println("Publisher: " +
resultSet.getString("Name"));
  }
```

5: Handling NULL values:

- IS NULL and IS NOT NULL handle rows with missing data
- Code example: Retrieves all books that don't specify a price
- IS NOT NULL would retrieve all rows where Price column has a value
- Use IS NULL and IS NOT NULL to ensure data completeness

```
ResultSet resultSet = statement.executeQuery(
  "SELECT * FROM Books WHERE Price IS NULL");
while (resultSet.next()) {
  System.out.println("Title: " +
resultSet.getString("Title"));
}
```

Note :

Common Methods that are used with Statements :

◆ Common Methods:

Method	Description
`executeQuery(String sql)`	Executes a SELECT query. Returns `ResultSet`.
`executeUpdate(String sql)`	Executes INSERT, UPDATE, DELETE. Returns number of affected rows.
`execute(String sql)`	Executes any SQL. Returns boolean indicating if result is `ResultSet`.

🔍 The 3 main methods in `java.sql.Statement`

Method	Used For	Returns	Example
`executeQuery(String sql)`	For **SELECT** statements (that return a ResultSet)	`ResultSet`	`ResultSet rs = stmt.executeQuery("SELECT student");`
`executeUpdate(String sql)`	For **INSERT**, **UPDATE**, **DELETE**, or DDL (CREATE, DROP, ALTER) statements	`int` (number of rows affected)	`int rows = stmt.executeUpdate("UPDATE SET sname='Raj' WHERE sid=`
`execute(String sql)`	For **any SQL** (when you don't know if it returns data or not)	`boolean` (`true` = ResultSet, `false` = update count or no result)	`boolean hasResult = stmt.execute("SELECT * FRO student");`

1 `executeQuery()`

- Only for **queries that return data** (SELECT).
- Returns a `ResultSet` you can iterate over.
- If you use it for an INSERT/UPDATE/DELETE, it throws an exception.

Example:

```java
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM student");

while (rs.next()) {
    System.out.println(rs.getString("sname"));
}
```

2 `executeUpdate()`

- For **data modification** or **DDL** (INSERT, UPDATE, DELETE, CREATE, DROP, etc.)
- Returns an integer = number of rows affected.
- Returns **0** for DDL (CREATE TABLE, DROP TABLE).

Example:

```java
int rows = stmt.executeUpdate("UPDATE student SET sname = 'Ravi' WHERE sid = 1");
System.out.println(rows + " row(s) updated.");
```

3 `execute()`

- Can handle **any SQL** (both SELECT and UPDATE/DDL).
- Returns `true` if there's a `ResultSet` (like SELECT),
  or `false` if it was an update (INSERT/UPDATE/DELETE).
- You can then use:
  - `stmt.getResultSet()` to retrieve data
  - `stmt.getUpdateCount()` to get affected rows

Example:

```java
boolean hasResult = stmt.execute("SELECT * FROM student");

if (hasResult) {
    ResultSet rs = stmt.getResultSet();
    while (rs.next()) {
        System.out.println(rs.getString("sname"));
    }
} else {
    int count = stmt.getUpdateCount();
    System.out.println("Rows affected: " + count);
}
```

Prepared and Callebale Statements :

What you will learn

- Describe the purpose and advantages of PreparedStatement and CallableStatement objects in JDBC
- Explain how to securely execute parameterized queries using PreparedStatement
- Explain how to call stored procedures with CallableStatement
- Describe the differences between Statement, PreparedStatement, and CallableStatement
- Describe common use cases, best practices, and error-handling techniques

1: Prepared Statement:

A PreparedStatement is a precompiled SQL statement(object) in JDBC that allows you to execute parameterized queries.

Prepared Statements allows you to use placeholders in queries, instead of allowing direct answers .

It acts as an interface provided by the JDBC API .

It's an **interface** provided by the JDBC API:

```
java

java.sql.PreparedStatement
```

Advatgaes of Prepared Statements :

Advantages:

- Prevents SQL injection
- Improves performance
- Enables easy reuse

Prepared statements are used when:

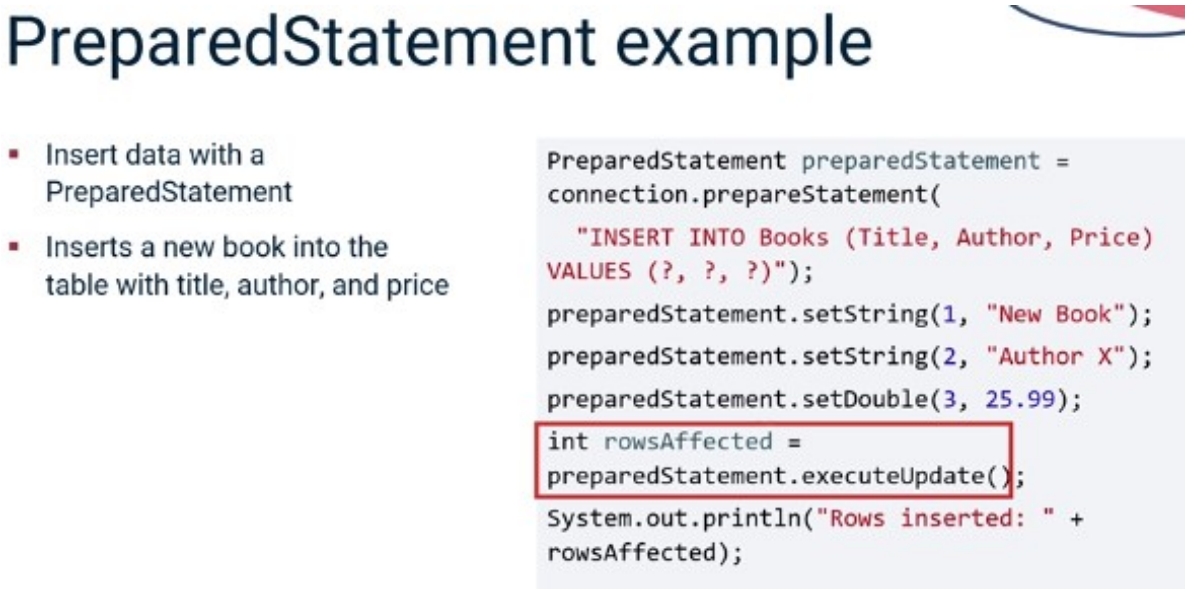
- You need to run the **same SQL query multiple times** with different values.
- You want to **avoid SQL injection attacks**.
- You need **better performance** through precompilation.

⚙ How PreparedStatement Works

1. You define a SQL query with **placeholders** (`?`) for parameters.
2. The query is **compiled once** by the database.
3. You **bind values** to the placeholders using `setXXX()` methods.
4. You **execute** the query as many times as needed, changing only the parameter values.

Example:

1. Prepared Statement for INSERT opeartion:



2. Prepared Statement for UPDATE opeartions:

PreparedStatement example (continued)

- Update data with a PreparedStatement
- Updates the price of the book
- Reusing query structure improves efficiency

```
PreparedStatement preparedStatement =
connection.prepareStatement(
  "UPDATE Books SET Price = ? WHERE Title = ?");
preparedStatement.setDouble(1, 30.99);
preparedStatement.setString(2, "New Book");
int rowsAffected =
preparedStatement.executeUpdate();
System.out.println("Rows updated: " +
rowsAffected);
```

where we ruse the same code and query of INSERT to UPDATE also.

3. Prepared Statement for SELECT operation with WHERE condition.

- Query data with a PreparedStatement
- SELECT query retrieves data, and parameters make the query dynamic
- Iterating over the ResultSet allows you to process each row returned

```
PreparedStatement preparedStatement =
connection.prepareStatement(
  "SELECT * FROM Books WHERE Author = ?");
preparedStatement.setString(1, "Author X");
ResultSet resultSet = preparedStatement.executeQuery();
while (resultSet.next()) {
  System.out.println("Title: " +
resultSet.getString("Title"));
  System.out.println("Price: " +
resultSet.getDouble("Price"));
}
```

Note :

Auto Generated Key :

When you insert a new row into a database table, the database can automatically generate a key value for a column (usually the primary key), means it can assign a auto genereted value to particualr column (of Primary key).

For example, many databases have columns defined as AUTO_INCREMENT (MySQL) or

IDENTITY (SQL Server).

```sql
CREATE TABLE employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50),
    department VARCHAR(50)
);
```

When you run this SQL:

```sql
INSERT INTO employees (name, department) VALUES ('John', 'Sales');
```

The database automatically generates a value for `id` — say, `101`.

In JDBC, you can retrieve that **auto-generated key** (like `id = 101`) using a special feature of the `PreparedStatement` object.

How to Retrieve Auto-Generated Keys

When you create a `PreparedStatement`, you can ask JDBC to **return generated keys** after executing an insert.

There are two main steps:

Step 1: Create PreparedStatement with Generated Keys Option

You do this using:

```java
Connection con = DriverManager.getConnection(...);

String sql = "INSERT INTO employees (name, department) VALUES (?, ?)";
PreparedStatement pstmt = con.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
```

The constant `Statement.RETURN_GENERATED_KEYS` tells the database you want the automatically generated keys back.

Step 2 : PreparedStatement are used to retrieve Auto-Generated Keys

After executing the insert, call:

```java
pstmt.executeUpdate();
ResultSet rs = pstmt.getGeneratedKeys();
```

Using a PreparedStatement object to retrieve auto-generated keys

- Insert data and retrieve generated key

```
PreparedStatement preparedStatement =
connection.prepareStatement(
  "INSERT INTO Books (Title, Author, Price) VALUES (?, ?, ?)",
  Statement.RETURN_GENERATED_KEYS);
preparedStatement.setString(1, "Generated Key Book");
preparedStatement.setString(2, "Author Y");
preparedStatement.setDouble(3, 30.00);
preparedStatement.executeUpdate();
ResultSet generatedKeys = preparedStatement.getGeneratedKeys();
if (generatedKeys.next()) {
  int newBookID = generatedKeys.getInt(1);
  System.out.println("Generated Book ID: " + newBookID);
}
```

2: Callable Statements :

A CallableStatement is a JDBC interface used to call stored procedures and functions that are stored in a database.

It is part of the java.sql package and extends the PreparedStatement interface.

That means it supports parameters — both input and output — and can be precompiled for efficiency.

Interface:

```
java

java.sql.CallableStatement
```

CallableStatement example

- Call a stored procedure with an input parameter

```
CREATE PROCEDURE GetBooksByAuthor(IN authorName VARCHAR(255))
BEGIN
  SELECT Title, Price FROM Books WHERE Author = authorName;
END;
CallableStatement callableStatement = connection.prepareCall("{CALL
GetBooksByAuthor(?)}");
callableStatement.setString(1, "Author X");
ResultSet resultSet = callableStatement.executeQuery();
while (resultSet.next()) {
  System.out.println("Title: " + resultSet.getString("Title"));
  System.out.println("Price: " + resultSet.getDouble("Price"));
}
```

⚙️ What is a Stored Procedure?

A **stored procedure** is a **predefined SQL program** stored in the database.
It can perform operations like inserting, updating, or querying data.

Example (MySQL):

```sql
CREATE PROCEDURE getEmployeeDetails(IN empId INT)
BEGIN
    SELECT name, department FROM employees WHERE id = empId;
END;
```

You can call this procedure using a `CallableStatement` in Java.

: where procedure is said as a Function

Callable Statement Example:

sql:

```sql
CREATE PROCEDURE addEmployee(IN id INT, IN name VARCHAR(50), IN dept VARCHAR(50))
BEGIN
    INSERT INTO employees (emp_id, emp_name, department) VALUES (id, name, dept);
END;
```

java:

```java
try {
    // Step 1: Load driver and connect
    Connection con = DriverManager.getConnection(
            url: "jdbc:mysql://localhost:3306/mydb", user: "user", password: "password");

    // Step 2: Create CallableStatement
    CallableStatement cstmt = con.prepareCall( sql: "{call addEmployee(?, ?, ?)}");

    // Step 3: Set input parameters
    cstmt.setInt( parameterIndex: 1, x: 101);
    cstmt.setString( parameterIndex: 2, x: "John Doe");
    cstmt.setString( parameterIndex: 3, x: "Sales");

    // Step 4: Execute
    cstmt.execute();

    System.out.println("Employee added successfully!");

    // Step 5: Close resources
    cstmt.close();
    con.close();

} catch (SQLException e) {
    e.printStackTrace();
}
```

Calleable Statement for finding the type of the Parameter :

The `registerOutParameter()` method is used in **JDBC CallableStatements** to **register an output parameter** that a stored procedure will return.

It's part of the `java.sql.CallableStatement` interface.

⚙️ Why It's Needed

When you call a **stored procedure** that has **OUT** or **INOUT** parameters,
you must **tell JDBC in advance** which parameters will receive data *from* the database.
That's exactly what `registerOutParameter()` does — it lets JDBC know to expect an output value for that parameter.

java.sql.Types.Double --> is responsible for finding the Type of Double .

✅ Example

Suppose you have this stored procedure in SQL:

```sql
CREATE PROCEDURE getBonus(IN empId INT, OUT bonus DECIMAL(10,2))
BEGIN
    SELECT salary * 0.1 INTO bonus FROM employees WHERE id = empId;
END;
```

java:

```java
try {
    Connection con = DriverManager.getConnection(
            url: "jdbc:mysql://localhost:3306/mydb", user: "user", password: "password");

    CallableStatement cstmt = con.prepareCall( sql: "{call getBonus(?, ?)}");

    // Set input parameter (IN)
    cstmt.setInt( parameterIndex: 1, x: 101);

    // Register output parameter (OUT)
    cstmt.registerOutParameter( parameterIndex: 2, Types.DECIMAL);

    // Execute stored procedure
    cstmt.execute();

    // Retrieve output value
    double bonus = cstmt.getDouble( parameterIndex: 2);
    System.out.println("Employee Bonus: " + bonus);

    cstmt.close();
    con.close();

} catch (SQLException e) {
    e.printStackTrace();
}
```

💡 How It Works

1. You tell JDBC which parameters are **OUT** parameters using `registerOutParameter()`.
2. When the stored procedure runs, the database fills those OUT parameters with values.
3. After execution, you can **read** those values using `getXXX()` methods (e.g. `getInt()`, `getString()` `getDouble()`).

```
CallableStatement cstmt = con.prepareCall("{call getBonus(?, ?)}");
```

- Creates a **CallableStatement** object for calling a stored procedure.
- The SQL string inside `"{}"` follows the syntax for calling procedures:
  - `{call procedure_name(?, ?, ...)}`
- Here, `getBonus` is the stored procedure name, and `(?, ?)` are its parameters:
  - The **first** `?` is an **IN parameter** (input to the procedure).
  - The **second** `?` is an **OUT parameter** (output returned by the procedure).

```
cstmt.setInt(1, 101);
```

- Sets the **value** for the **first parameter** (`?`) of the procedure.
- The index `1` refers to the **first placeholder**.
- So we are passing the value `101` as the **employee ID** (`empId`) input to the procedure.

```
cstmt.registerOutParameter(2, Types.DECIMAL);
```

- Registers the **second parameter** (`?`) as an **OUT parameter**.
- `2` → position of the second placeholder.
- `Types.DECIMAL` → specifies the SQL data type of the output (in this case, a decimal value for `bonus`)
- This tells JDBC: "Expect a decimal number from this parameter after executing the procedure."

```
cstmt.execute();
```

- Executes the **stored procedure** on the database.
- The database:
  1. Receives the employee ID (`101`).
  2. Runs the `getBonus` procedure.
  3. Calculates the bonus (e.g., 10% of salary).
  4. Returns that bonus value into the OUT parameter.

```
double bonus = cstmt.getDouble(2);
```

- Retrieves the **value** from the **second parameter** (the OUT parameter).
- The database has placed the bonus value there after execution.
- `getDouble(2)` reads it and stores it in the Java variable `bonus`.