Credit: Prepared by: Shagun Gujarathi

- **What is Java?** – A platform-independent, object-oriented programming language.
- **Why Java?** – Simple, secure, robust, and platform-independent.
- **Predecessor of Java:** – **C++**
- **Successor of Java:** – **Kotlin**
- **Who discovered Java?** – **James Gosling**
- **Year of discovery:** – **1995**

- **JDK (Java Development Kit)**
  - **Definition:** A software development kit used to **write, compile, and run Java programs**.
  - **Consists of:**
    - JRE (Java Runtime Environment)
    - Development Tools: `javac` (compiler), `java` (launcher), debugger, etc.
    - Java APIs and Libraries
  - **Architecture:**
    - Development Tools → Used to write & compile code
    - JRE → Provides the environment to run compiled code
    - Java APIs → Provide pre-defined functions
  - **Real-Time Example:**
    When a developer writes and compiles Java code in an IDE like IntelliJ or Eclipse, the JDK is required to develop and run the program.

- **JRE (Java Runtime Environment)**
  - **Definition:** Provides the **runtime environment** needed to run Java applications.
  - **Consists of:**
    - JVM (Java Virtual Machine)
    - Core Libraries (Java Class Libraries)
    - Runtime Libraries
  - **Architecture:**
    - Class Loader → Loads `.class` files
    - JVM → Executes bytecode
    - Libraries → Support runtime functionalities
  - **Real-Time Example:**
    When you install and run a Java-based app like Minecraft, the JRE ensures it works smoothly on your device.

- **JVM (Java Virtual Machine)**
  - **Definition:** The engine that **executes Java bytecode**, making Java platform-independent.
  - **Consists of:**
    - Class Loader Subsystem → Loads class files
    - Runtime Data Areas → Stack, Heap, Method Area
    - Execution Engine → Executes bytecode

        - Native Method Interface (JNI) → Interacts with native code (C/C++)
    - **Architecture:**
        - Class Loader → Runtime Data Areas → Execution Engine → Native Interface
        - Converts bytecode → Machine code → Executes
    - **Real-Time Example:**
      When you click "Run" on a Java program, the JVM takes the compiled bytecode and executes it on your machine, whether it's Windows, Linux, or macOS.

- **Simple Analogy (Book Example):**
    - **JDK:** Like an **author's toolkit** (pen, paper, dictionary) to write and publish a book (develop code).
    - **JRE:** Like a **reader's kit** (just the book and reading glasses) to read the book (run code).
    - **JVM:** Like a **translator** who translates the book into a language the reader understands (convert bytecode to machine code).

Class, Object, and OOPs in Java

1. Class

Definition: A **blueprint** or **template** for creating objects. It defines properties (variables) and behaviors (methods).

Syntax:

```
class Car {           // Class definition
    String color;      // Property
    int speed;         // Property

    void drive() {     // Method
        System.out.println("Car is driving");
    }
}
```

Real-Time Example:
Think of a **"Car"** class. It defines the general features of all cars, like color, speed, and the ability to drive.

How to Achieve:

- Use the `class` keyword to define a class.
- Define properties (fields) and behaviors (methods) inside the class.

2. Object

Definition: An **instance** of a class. It represents real-world entities with specific values for properties.

Syntax:

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();     // Creating an object
        myCar.color = "Red";       // Assigning property
```

```
        myCar.speed = 120;          // Assigning property
        myCar.drive();              // Calling method
    }
}
```

Real-Time Example:
If "**Car**" is the class, then "**myCar**" is an object. It's a specific car with properties like "Red" color and speed "120 km/h".

How to Achieve:

- Create objects using the `new` keyword.
- Access properties and methods using the dot (`.`) operator.

---

3. OOPs (Object-Oriented Programming) Concepts

- **Definition:** A programming paradigm based on the concept of **objects**. It helps in organizing code for better **reusability**, **scalability**, and **maintenance**.

Key Principles of OOP (4 Pillars):

---

1. Encapsulation

- **Definition:** Binding data (variables) and methods together, restricting direct access to data for security.
- **How to Achieve:**
  - Declare variables as `private`.
  - Provide `getter` and `setter` methods to access and modify the data.
  - **Syntax:**

```
class Person {
    private String name;     // Encapsulated field

    public String getName() { return name; }   // Getter
    public void setName(String name) { this.name = name; }   //
Setter
}
```

- **Real-Time Example:**
  In banking apps, your account balance is private. You can check or update it only through secure methods (getters/setters).

---

2. Abstraction

- **Definition:** Hiding complex implementation details and showing only the essential features to the user.
- **How to Achieve:**
  - Use **abstract classes** and **interfaces**.
  - Define abstract methods that subclasses must implement.
  - **Syntax:**

```
abstract class Vehicle {
    abstract void start();   // Abstract method
}

class Bike extends Vehicle {
    void start() {
        System.out.println("Bike starts with a key");
    }
}
```

- **Real-Time Example:**
  A **car driver** uses the steering wheel without knowing the internal mechanism of how it works.

---

3. Inheritance

- **Definition:** The process where one class **inherits** properties and methods from another class, promoting **code reusability**.
- **Types of Inheritance:**
  - **Single Inheritance:** One class inherits from another.
  - **Multilevel Inheritance:** A class inherits from a derived class.
  - **Hierarchical Inheritance:** Multiple classes inherit from one base class.
    (*Note: Java doesn't support multiple inheritance with classes but supports it with interfaces.*)
- **How to Achieve:**
  - Use the `extends` keyword for classes.
  - Use the `implements` keyword for interfaces.
  - **Syntax:**

```
class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}

class Car extends Vehicle {   // Inheritance using 'extends'
    void display() {
        System.out.println("Car is displayed");
    }
}
```

- **Real-Time Example:**
  A **"Car"** class can inherit common properties from a **"Vehicle"** class.

---

4. Polymorphism

- **Definition:** The ability of a method to **perform different tasks** based on the context, promoting **flexibility** in code.
- **Types of Polymorphism:**
  - **Compile-time (Static) Polymorphism:** Achieved through **method overloading** (same method name, different parameters).
  - **Run-time (Dynamic) Polymorphism:** Achieved through **method overriding** (subclass provides its own implementation of a superclass method).
- **How to Achieve:**
  - **Method Overloading:** Define multiple methods with the same name but different parameters.
  - **Method Overriding:** Override methods in child classes using `@Override`.
  - **Syntax:**

```
// Compile-time Polymorphism (Method Overloading)
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
    int add(int a, int b, int c) {  // Overloaded method
        return a + b + c;
    }
}

// Run-time Polymorphism (Method Overriding)
class Animal {
```

```
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

- **Real-Time Example:**
  - **Overloading:** A **printer** can print documents, images, PDFs using the same `print()` method but with different data formats.
  - **Overriding:** The `sound()` method behaves differently for `Dog` and `Cat` classes.

🎯 Summary of OOPs Concepts:

OOP Principle	How to Achieve	Real-Time Example
Encapsulation	Private variables + Getters/Setters	Bank account security
Abstraction	Abstract classes, Interfaces	Car steering system
Inheritance	extends (for classes), implements (for interfaces)	Car inherits features from Vehicle
Polymorphism	Method Overloading & Overriding	Printer handling different data formats

---

Design Patterns in Software Development

✅ What Are Design Patterns?

- Design patterns are **proven solutions** to common software design problems.
- They provide **best practices** for writing clean, reusable, and maintainable code.
- **Purpose:** Improve code flexibility, scalability, and reduce redundancy.

🔍 Types of Design Patterns

- **Creational Patterns:** Deals with object creation.
- **Structural Patterns:** Focuses on object composition.
- **Behavioral Patterns:** Manages communication between objects.

🏗️ 1. Creational Design Patterns

- **Purpose:** Control the creation process of objects.
- **Examples:**
    - **Singleton Pattern:** Ensures a class has only one instance.
        - *Real-Time Example:* A **logging service** where only one instance logs throughout the app.
    - **Factory Pattern:** Creates objects without specifying the exact class.
        - *Real-Time Example:* A **payment gateway** system deciding whether to process a **credit card** or **UPI** payment dynamically.
    - **Builder Pattern:** Builds complex objects step-by-step.
        - *Real-Time Example:* **Pizza order system** where you select base, toppings, and sauce separately.

🧩 2. Structural Design Patterns

- **Purpose:** Simplify the structure of code by organizing classes and objects.
- **Examples:**
    - **Adapter Pattern:** Converts one interface into another for compatibility.
        - *Real-Time Example:* **Mobile charger adapter** connecting incompatible devices.
    - **Facade Pattern:** Provides a simple interface to complex subsystems.
        - *Real-Time Example:* A **unified dashboard** controlling different modules like reports, users, and analytics.

🔄 3. Behavioral Design Patterns

- **Purpose:** Manage how objects interact and communicate.
- **Examples:**
    - **Strategy Pattern:** Chooses an algorithm at runtime based on conditions.
        - *Real-Time Example:* A **navigation app** selecting the fastest or shortest route dynamically.
    - **Observer Pattern:** Notifies multiple objects when a state changes.
        - *Real-Time Example:* **YouTube notifications** alerting subscribers when a new video is uploaded.

SOLID Principles in the Project

✅ What Are SOLID Principles?

SOLID is an acronym for five design principles that help in building **robust, maintainable, and scalable software**:

- **S - Single Responsibility Principle (SRP)**
- **O - Open/Closed Principle (OCP)**
- **L - Liskov Substitution Principle (LSP)**
- **I - Interface Segregation Principle (ISP)**
- **D - Dependency Inversion Principle (DIP)**

By following SOLID principles, the project became:

- **More maintainable**
- **Easier to scale**
- **Less prone to bugs** when adding new features ✅

Java 8 vs Java 11 vs Java 17 vs Java 21 (Key Differences)

Feature	Java 8 (2014)	Java 11 (2018) [LTS]	Java 17 (2021) [LTS]	Java 21 (2023) [LTS]
Programming Style	Lambda Expressions (functional style)	var in Lambda	Pattern Matching (instanceof)	Record Patterns, String Templates
Collections & Data Handling	Streams API	Files.readString()	Enhanced switch expressions	Unnamed Classes, Simplified Code
Date & Time API	New Date & Time API (java.time)	—	—	—
HTTP Client	Basic HttpURLConnection	New HTTP Client API	—	—
String Enhancements	Basic String Methods	isBlank(), lines(), strip(), repeat()	Text Blocks (""")	String Templates
OOP Enhancements	Default & Static Methods in Interfaces	—	Sealed Classes	Virtual Threads (Improved Concurrency

Feature	Java 8 (2014)	Java 11 (2018) [LTS]	Java 17 (2021) [LTS]	Java 21 (2023) [LTS]
Null Handling	Optional Class	—	—	—
Performance & Memory	Standard GC	Improved Garbage Collection	Strong Encapsulation of Internal APIs	Enhanced Garbage Collection
Removed Features	—	Removed Java EE Modules (JAXB, JAX-WS)	—	—
Concurrency	Standard Threads	—	—	Virtual Threads for Better Performance

---

🚀 Summary:

- **Java 8:** Functional programming (Lambda, Streams).
- **Java 11:** Better HTTP handling, String API improvements.
- **Java 17:** OOP enhancements with Sealed Classes, Pattern Matching.
- **Java 21:** Focus on concurrency (Virtual Threads), cleaner syntax.

Here's an expanded **Multithreading & Core Java Concepts Summary** with additional topics, keeping it concise and Word-friendly:

Multithreading & Core Java Concepts

Concept	Description	Syntax/Example	Real-Time Example
Thread	Smallest unit of a process.	`Thread t = new Thread(); t.start();`	Downloading files while browsing.
Runnable Interface	Defines task logic separately from thread.	`class Task implements Runnable { public void run() {} }`	Sending emails in the background.

Concept	Description	Syntax/Example	Real-Time Example
run() vs start()	run() runs on the current thread; start() creates a new thread.	thread.start(); // vs thread.run();	start() is used for parallel execution.
Thread Lifecycle	New → Runnable → Running → Blocked → Terminated	Handled by JVM.	Video buffering with pausing/resuming.
Synchronization	Prevents race conditions in shared resources.	synchronized(obj) { // critical section }	Bank transactions to avoid double deductions.
Deadlock	Circular dependency causing threads to block forever.	Improper nested locks.	Traffic jam without signals.
Volatile Keyword	Ensures thread visibility of variables.	volatile boolean flag = true;	Real-time game status flag.
Thread Pool (Executor)	Manages threads efficiently.	ExecutorService ex = Executors.newFixedThreadPool(5);	Handling API requests simultaneously.
Callable & Future	Executes tasks and returns results.	Future<Integer> result = executor.submit(task);	Fetching stock prices concurrently.
Join	Waits for one thread to finish.	thread.join();	Uploading files before sending confirmation.

Concept	Description	Syntax/Example	Real-Time Example
Sleep	Pauses execution for a set time.	Thread.sleep(1000);	Auto-logout timers.

Core Java Concepts

Concept	Description	Syntax/Example	Real-Time Example
Serialization	Converts objects to byte stream.	ObjectOutputStream oos = new ObjectOutputStream();	Saving user sessions.
Deserialization	Converts byte stream back to object.	ObjectInputStream ois = new ObjectInputStream();	Restoring app data after restart.
Transient Keyword	Prevents fields from being serialized.	transient int tempData;	Ignoring sensitive data like passwords.
StringBuffer	Mutable, thread-safe string operations.	StringBuffer sb = new StringBuffer("Hello");	Logging in multi-threaded apps.
StringBuilder	Mutable, faster (not thread-safe).	StringBuilder sb = new StringBuilder("Hi");	Building strings in single-threaded apps.
Immutable Strings	Once created, cannot be changed.	String s = "Test"; s.concat("123");	Storing constant values like config keys.
Mutable Objects	Can be changed	List<String> list = new ArrayList<>();	Modifying shopping cart items.

Concept	Description	Syntax/Example	Real-Time Example
	after creation.		
Transaction Management	Ensures data consistency in DB operations.	@Transactional (Spring)	Bank money transfer (debit/credit both succeed or fail).
Lambda Expressions	Enables functional programming.	(a, b) -> a + b;	Filtering data in collections.
Functional Interface	Interface with a single abstract method.	@FunctionalInterface	Runnable, Comparator implementations.
Optional Class	Handles null values gracefully.	Optional.ofNullable(value)	Preventing NullPointerException.

🚀 Key Takeaways:

- **Multithreading:** Boosts performance with parallel task execution.
- **Synchronization & Volatile:** Manage data consistency in concurrent environments.
- **Core Java:** Serialization, String handling, and transaction management ensure robust applications.

Advanced Java Concepts

Concept	Description	Syntax/Example	Real-Time Example
Abstract Class	Cannot be instantiated, used as a base for other	abstract class Vehicle { abstract void run(); }	Defining common properties of vehicles.

Concept	Description	Syntax/Example	Real-Time Example
	classes.		
Interface	Defines a contract with abstract methods.	interface Drivable { void drive(); }	Payment gateways implementing Payable.
Polymorphism	Performing actions in different ways using the same interface.	Vehicle v = new Car(); v.run();	Print functions handling different data types.
Encapsulation	Binding data and methods together, hiding internal details.	private int age; public void setAge(int a)	User data protection in banking apps.
Inheritance	Acquiring properties of parent class.	class Dog extends Animal {}	Employee class inheriting Person details.
Exception Handling	Manages runtime errors to prevent crashes.	try { } catch (Exception e) { } finally { }	Handling invalid user inputs in forms.
Checked Exception	Caught at compile-time.	throws IOException	File reading errors.
Unchecked Exception	Occurs at runtime.	int a = 5 / 0;	NullPointerException when accessing null.
Generics	Allows code reusability with type safety.	List<String> list = new ArrayList<>();	Type-safe collections in data structures.

Concept	Description	Syntax/Example	Real-Time Example
Collections Framework	Provides data structures like List, Set, Map.	List<Integer> nums = new ArrayList<>();	Storing items in an e-commerce cart.
HashMap vs HashSet	HashMap stores key-value pairs; HashSet stores unique values.	Map<Key, Value> vs Set<Value>	Employee IDs in HashSet, data caching with HashMap.
Comparable vs Comparator	Sorting objects naturally vs custom sorting logic.	compareTo() vs compare()	Sorting employee data by name or salary.
Enum	Defines fixed constants.	enum Day { MON, TUE, WED }	Days of the week, status codes.
Static Keyword	Belongs to the class, not instances.	static int count;	Common counter for all objects.
Final Keyword	Prevents modification (variables, methods, classes).	final int x = 100;	Defining constant values like PI.
This Keyword	Refers to the current object.	this.name = name;	Resolving variable shadowing in constructors.
Super Keyword	Refers to the parent class.	super(); or super.method()	Calling parent class constructor/method.

Concept	Description	Syntax/Example	Real-Time Example
Instanceof Operator	Checks object type at runtime.	if (obj instanceof String)	Type checking before casting.
Immutable Class	Class whose objects cannot be modified after creation.	final class Person { private final String name; }	String class is immutable.
Reflection API	Inspects classes, methods, and fields at runtime.	Class<?> cls = Class.forName("ClassName");	Dynamic loading of plugins or modules.
Annotations	Provides metadata about code.	@Override, @Deprecated, @FunctionalInterface	Marking deprecated methods in APIs.
Java Streams (Java 8)	Processes collections in a functional style.	list.stream().filter(e -> e > 10).collect(Collectors.toList());	Filtering data in large datasets.
Optional (Java 8)	Handles null values gracefully to prevent exceptions.	Optional.ofNullable(value)	API responses to avoid NullPointerException.
Date & Time API (Java 8)	Improved date-time handling with LocalDate, LocalTime.	LocalDate.now(); LocalDate.of(2024, 2, 9);	Event scheduling apps.
Var Keyword	Local variable	var name = "Shagun";	Reducing boilerplate code.

Concept	Description	Syntax/Example	Real-Time Example
(Java 10)	type inference.		
Records (Java 14)	Immutable data classes with less boilerplate.	record User(String name, int age) {}	DTOs in APIs.
Sealed Classes (Java 15)	Restricts class inheritance.	sealed class Shape permits Circle, Square {}	Modeling restricted hierarchies.
Pattern Matching (Java 16)	Simplifies type casting in conditions.	if (obj instanceof String s) { System.out.println(s); }	Simplifying conditional logic.

🚀 Key Takeaways:

- **OOP Concepts:** Drive reusability, modularity, and scalability.
- **Advanced Features:** Streams, Optionals, and Reflection improve modern Java development.
- **Best Practices:** Encapsulation, immutability, and annotations ensure clean, maintainable code.

🚀 Comprehensive Java Concepts Summary

Concept	Description	Example/Use Case
== vs .equals()	== compares references, .equals() compares values.	str1.equals(str2) for string content check.
final Keyword	Prevents modification of variables, methods, classes.	final int x = 10; (cannot reassign).
finally Block	Executes code after try-catch regardless of	Closing DB connections in finally.

Concept	Description	Example/Use Case
	exceptions.	
finalize() Method	Called before garbage collection for cleanup (deprecated).	Releasing resources (rarely used now).
Checked Exceptions	Checked at compile-time; must be handled with try-catch.	IOException, SQLException.
Unchecked Exceptions	Runtime exceptions; not required to be caught explicitly.	NullPointerException, ArrayIndexOutOfBoundsException.
volatile Keyword	Ensures visibility of changes to variables across threads.	Shared flags in multi-threaded apps.
transient Keyword	Prevents variable serialization.	Hiding passwords during serialization.
synchronized Keyword	Controls thread access to critical sections.	Thread-safe method implementation.
Callable & Future	Asynchronous programming with results.	Fetching data from APIs concurrently.
Lambda Expressions (Java 8)	Enables functional programming.	list.forEach(e -> System.out.println(e));
Functional Interfaces	Interface with a single abstract method.	@FunctionalInterface for lambda usage.
Method References (Java 8)	Shorthand for lambda expressions.	System.out::println
Predicate, Consumer, Supplier	Functional interfaces for data processing.	Filtering with Predicate<Employee>.

Concept	Description	Example/Use Case
Serialization/Deserialization	Convert objects to byte streams and vice versa.	Saving user sessions or data transfer.
JVM Memory Model	Heap, Stack, Metaspace, etc.	Optimizing JVM performance.
Garbage Collection (GC)	Automatic memory management in JVM.	JVM tuning for large-scale apps.
Soft, Weak, Phantom References	Manage memory efficiently with different reference types.	Caching mechanisms.
Dynamic Binding (Late Binding)	Method resolution at runtime.	Overridden methods in polymorphism.
Static Binding (Early Binding)	Method resolution at compile-time.	Method overloading.
Assertions	Debugging tool to test assumptions.	assert age > 18; for validation.
JIT Compiler	Optimizes bytecode at runtime for faster execution.	Performance improvement in JVM.
Transaction Management (Spring)	Ensures data consistency during DB operations.	@Transactional for atomic DB transactions.
Dependency Injection (DI)	Manages object dependencies automatically.	@Autowired in Spring Boot.
Bean Lifecycle (Spring)	Creation, initialization, destruction of beans.	Using @PostConstruct, @PreDestroy.
AOP (Aspect-Oriented Programming)	Separates cross-cutting concerns like logging and security.	@Aspect for logging API requests.

🚀 Java Concepts: Stream API, Collections, Garbage Collection

📊 1. Stream API (Java 8)

Aspect	Details
Purpose	Process collections of data in a functional way (filtering, mapping, sorting).
Basic Syntax	list.stream().filter(e -> e > 10).collect(Collectors.toList());
Real-Time Example	Filtering employees with salary > 50,000:employees.stream().filter(emp -> emp.getSalary() > 50000).collect(Collectors.toList());
Key Operations	filter(), map(), sorted(), collect(), forEach(), reduce()

📦 2. Collections Framework

Type	Description	Real-Time Example
List	Ordered, allows duplicates.	List<String> names = new ArrayList<>(); (Employee Names)
Set	Unordered, no duplicates.	Set<Integer> ids = new HashSet<>(); (Unique IDs)
Queue	FIFO (First In, First Out).	Queue<String> tasks = new LinkedList<>(); (Task Scheduling)
Map	Key-value pairs.	Map<Integer, String> empMap = new HashMap<>(); (Employee ID to Name)

🔍 3. Differences Between Collections

Collection	Implementation	Key Differences
List	ArrayList vs LinkedList	ArrayList: Fast for random access, slow for insertion/deletion.LinkedList: Fast insertion/deletion, slower random access.
Set	HashSet vs TreeSet	HashSet: Unordered, faster (uses hash table).TreeSet: Sorted order, slower (uses Red-Black tree).
Queue	PriorityQueue vs	PriorityQueue: Orders elements based on

Collection	Implementation	Key Differences
	Deque	priority.Deque: Double-ended queue, allows insertion/removal from both ends.
Map	HashMap vs TreeMap vs LinkedHashMap	HashMap: Unordered, fast performance.TreeMap: Sorted by keys.LinkedHashMap: Maintains insertion order.

🗑️ 4. Garbage Collection (GC)

Aspect	Details
Purpose	Automatic memory management in JVM.
Why We Use It?	Prevents memory leaks, improves performance by freeing unused objects.
When It Happens?	Triggered automatically when JVM detects low memory or manually using System.gc().
Types of GC	1️⃣ **Serial GC** (Simple, single-threaded) 2️⃣ **Parallel GC** (Multi-threaded, for high performance) 3️⃣ **CMS (Concurrent Mark-Sweep)** (Low-latency apps) 4️⃣ **G1 GC** (Optimized for large heaps, Java 9+)
Real-Time Example	In large-scale web applications to handle session data and temporary objects efficiently.

😊

Optional Additional Java Concepts

Concept	Description	Real-Time Example	Basic Syntax (if applicable)
Reflection API	Inspect or modify runtime behavior of classes, methods, and fields.	Frameworks like Hibernate, Spring (dynamic object creation).	Class<?> clazz = Class.forName("com.example.MyClass");Method method = clazz.getMethod("methodName");
Java Modules (Java 9+)	Modularize large applications for better maintainabi	Enterprise applications with independent modules.	module com.example.app { requires java.base; }

Concept	Description	Real-Time Example	Basic Syntax (if applicable)
	lity and performance.		
Concurrency Utilities	Manage complex multithreading tasks with synchronization tools.	Thread coordination in banking transactions.	CountDownLatch latch = new CountDownLatch(1);latch.await(); latch.countDown();
Reactive Programming (WebFlux)	Handle asynchronous data streams with non-blocking behavior.	Real-time applications like chat apps or stock monitoring.	Mono<String> mono = Mono.just("Hello");mono.subscribe(System.out::println);
Optional Class (Java 8)	Avoid NullPointerException by handling null values gracefully.	Handling optional user data in APIs.	Optional<String> opt = Optional.ofNullable(null);opt.orElse("Default Value");
Immutable Classes	Class whose objects cannot be modified after creation.	String class in Java is immutable.	final class Employee { private final String name; }
Autoboxing/ Unboxing	Automatic conversion between primitive types and wrapper classes.	Simplifies collection operations with primitives.	List<Integer> list = Arrays.asList(1, 2, 3);
Method References	Shorter syntax for calling methods in lambda	Stream operations like sorting or filtering.	List<String> names = List.of("A", "B");names.forEach(System.out::println);

Concept	Description	Real-Time Example	Basic Syntax (if applicable)
	expressions.		
Annotations	Metadata that provides information to the compiler or runtime.	@Override in method overriding, @Autowired in Spring.	@Override public String toString() { return "Example"; }
Functional Interfaces	Interfaces with a single abstract method used for lambda expressions.	Java 8 Streams, event listeners.	@FunctionalInterface interface MyFunc { void execute(); }
Wrapper Classes	Provides a way to use primitive data types as objects.	Collections that require objects (e.g., ArrayList<Integer>).	Integer num = Integer.valueOf(5);
Java Memory Model	Describes how threads interact through memory (Heap, Stack, Metaspace).	Memory management in high-concurrency apps.	*No specific syntax (conceptual understanding)*
Predicate & Consumer (Java 8)	Built-in functional interfaces used with Streams.	Filtering employee data in APIs.	Predicate<Integer> isPositive = num -> num > 0;Consumer<String> printer = System.out::println;
Typecasting	Converting one data type into another.	1 Implicit: int to double in	**Implicit:** int a = 10; double b = a;**Explicit:** double d = 9.7; int i = (int) d;

Concept	Description	Real-Time Example	Basic Syntax (if applicable)
	Two types: [1] **Implicit (Widening)** - Auto conversion (smaller to larger). [2] **Explicit (Narrowing)** - Manual conversion (larger to smaller).	calculations. [2] Explicit: Casting double to int in finance apps.	

Spring Framework Concepts

- **Spring Framework:** A lightweight, open-source framework for building enterprise Java applications, providing support for dependency injection, transaction management, and more.
- **Dependency Injection (DI):** Injects object dependencies automatically, reducing tight coupling.
- **Inversion of Control (IoC):** The framework controls object creation and lifecycle instead of the developer.
- **Bean:** An object managed by the Spring IoC container.
- **ApplicationContext:** Advanced IoC container for managing beans and resources.
- **Spring MVC:** A web framework following the Model-View-Controller pattern for creating web applications.
- **Service Layer:** Contains business logic and communicates between controllers and repositories.
- **Repository Layer:** Handles data access operations, usually with databases.
- **Transaction Management:** Ensures data consistency by managing database transactions.
- **AOP (Aspect-Oriented Programming):** Manages cross-cutting concerns like logging and security.
- **Event Handling:** Enables communication between different parts of an application using events.
- **Caching:** Improves performance by storing frequently accessed data temporarily.
- **External Configuration:** Manages app configurations through external files (like `.properties` files).
- **Profiles:** Allows setting up different configurations for different environments (dev, test, prod).

---

1. Dependency Injection (DI) in Spring

Definition:
Dependency Injection (DI) is a design pattern where the Spring container automatically injects dependencies into objects, reducing tight coupling and improving code manageability.

Types of Dependency Injection

1. Constructor Injection:

Dependencies are provided through the class constructor.

Example:

```
public class Service {
    private Repository repo;

    public Service(Repository repo) {  // Constructor
Injection
        this.repo = repo;
    }
}
```

Setter Injection:

Dependencies are injected via public setter methods.

Example:

```
public class Service {
    private Repository repo;

    public void setRepository(Repository repo) {  //
Setter Injection
        this.repo = repo;
    }
}
```

Field Injection:

Dependencies are injected directly into class fields using `@Autowired`.

Example:

```
@Autowired
private Repository repo;
```

✅ Key Points to Remember:

- **Constructor Injection** is preferred for mandatory dependencies (ensures immutability).
- **Setter Injection** is useful for optional dependencies.

- **Field Injection** reduces boilerplate code but is harder to test (not recommended for complex applications).

---

2. Inversion of Control (IoC):

A principle where the control of object creation and lifecycle is handled by the Spring container.

- **Types of IoC Containers:**
  - **BeanFactory:** Basic IoC container, lazy initialization (creates beans only when needed).
  - **ApplicationContext:** Advanced container, eager initialization (creates beans at startup) with extra features like event propagation, AOP support, etc.

---

3. MVC Architecture (Model-View-Controller):

A design pattern to separate application logic into three interconnected layers.

How MVC Works:

1. **Model:** Represents the data and business logic (e.g., database operations).
2. **View:** The user interface (e.g., HTML, JSP).
3. **Controller:** Handles user requests, processes data via the Model, and returns the View.

Flow in Spring MVC:

1. **Client Request** →
2. **DispatcherServlet (Front Controller)** →
3. **Controller (Handles logic)** →
4. **Service Layer (Business logic)** →
5. **DAO/Repository (Database interaction)** →
6. **Model (Data)** →
7. **View Resolver (Determines UI)** →
8. **Response to Client**

Spring Boot

- **Spring Boot:** A framework built on top of Spring to create stand-alone, production-ready applications with minimal configuration.
- **Why Use Spring Boot:** Simplifies development with auto-configuration, embedded servers, and easy dependency management.
- **Predecessor:** Spring Framework
- **Successor:** Latest version - Spring Boot 3.x with Spring Framework 6

Spring Framework vs Spring Boot

Aspect	Spring Framework	Spring Boot
Definition	A comprehensive framework for Java applications.	A tool built on top of Spring to simplify development.
Configuration	Requires manual XML or Java-based configurations.	Auto-configuration with minimal setup.
Deployment	Requires external servers like Tomcat.	Embedded servers (Tomcat, Jetty) for easy deployment.
Dependency Management	Manual handling of dependencies.	Uses starter POMs for easy dependency management.
Complexity	More boilerplate code and setup required.	Reduces boilerplate code, faster development.
Use Case	Suitable for complex, enterprise applications.	Best for microservices and REST API development.

Spring Boot Components, Architecture, and Annotations

🚀 Spring Boot Components:

- **Spring Boot Starter:** Pre-configured dependencies for quick setup.
- **Spring Boot Auto-Configuration:** Automatically configures beans based on project dependencies.
- **Spring Boot CLI:** Command-line tool for running and testing Spring Boot applications.
- **Spring Boot Actuator:** Provides production-ready features like monitoring and metrics.
- **Spring Initializr:** Web-based tool to generate Spring Boot project structures.

🏗️ Spring Boot Architecture:

1. Internal Architecture:

- **Core Container:** Manages dependency injection via ApplicationContext.
- **Auto-Configuration:** Detects project setup and applies default configurations.
- **Embedded Servers:** Uses Tomcat, Jetty, or Undertow to run apps without external servers.
- **Spring MVC:** Handles HTTP requests/responses, RESTful APIs.
- **Data Access Layer:** Integrates with JPA, JDBC, MongoDB, etc.

2. External Architecture:

- **Client Layer:** User interface or API client (like Angular, Postman).

- **Controller Layer:** Manages HTTP requests with `@RestController` and `@RequestMapping`.
- **Service Layer:** Business logic using `@Service`.
- **Repository Layer:** Database operations using `@Repository` (JPA, CRUD).
- **Database Layer:** SQL (MySQL, PostgreSQL) or NoSQL (MongoDB).

📌 Spring Boot Annotations

1. Core Annotations:

- **@SpringBootApplication:** Main entry point for Spring Boot apps (combines `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`).
- **@ComponentScan:** Scans specified packages for Spring components.
- **@Configuration:** Marks a class as a source of bean definitions.
- **@Bean:** Defines a bean to be managed by the Spring container.
- **@EnableAutoConfiguration:** Enables automatic configuration based on dependencies in the classpath.

2. RESTful API Annotations:

- **@RestController:** Simplifies creating RESTful web services (combines `@Controller` and `@ResponseBody`).
- **@GetMapping:** Maps HTTP GET requests to specific handler methods.
- **@PostMapping:** Maps HTTP POST requests to specific handler methods.
- **@PutMapping:** Maps HTTP PUT requests (for updating resources).
- **@PatchMapping:** Maps HTTP PATCH requests (for partial updates).
- **@DeleteMapping:** Maps HTTP DELETE requests to handler methods.
- **@RequestMapping:** General-purpose annotation for mapping requests to handler methods.
- **@RequestParam:** Extracts query parameters from the URL.
- **@PathVariable:** Binds URI template variables to method parameters.
- **@RequestBody:** Maps the HTTP request body to a Java object.
- **@ResponseBody:** Converts Java objects to JSON/XML responses.
- **@CrossOrigin:** Enables Cross-Origin Resource Sharing (CORS) for REST APIs.
- **@ResponseStatus:** Customizes HTTP response status for exceptions or methods.

3. Dependency Injection & Bean Scope:

- **@Autowired:** Automatically injects dependencies.
- **@Qualifier:** Specifies which bean to inject when multiple beans of the same type exist.
- **@Value:** Injects values from properties files.
- **@Scope:** Defines the scope of a bean (`singleton`, `prototype`, etc.).
- **@Lazy:** Delays bean initialization until it's needed.

4. AOP (Aspect-Oriented Programming) Annotations:

- **@Aspect:** Defines a class as an aspect for cross-cutting concerns (e.g., logging, security).
- **@Before, @After, @Around:** Apply actions before, after, or around method execution in AOP.
- **@EnableAspectJAutoProxy:** Enables support for handling components marked with `@Aspect`.

5. Scheduling & Async Annotations:

- **@Scheduled:** Schedules tasks to run at fixed intervals.
- **@EnableScheduling:** Enables scheduled tasks in Spring Boot.
- **@Async:** Executes methods asynchronously.
- **@EnableAsync:** Enables asynchronous processing.

6. Validation Annotations (Hibernate Validator):

- **@Valid / @Validated:** Validates request bodies or objects in APIs.
- **@NotNull, @Size, @Min, @Max, @Pattern:** Used for input validation.

7. Spring Security Annotations:

- **@EnableWebSecurity:** Enables Spring Security in the project.
- **@PreAuthorize, @PostAuthorize:** Adds method-level security based on roles.
- **@Secured:** Restricts access based on roles.
- **@RolesAllowed:** Specifies allowed roles for a method or class.

8. Transaction Management:

- **@Transactional:** Manages transactions automatically (rollbacks, commits).

9. Profile & Configuration:

- **@Profile:** Specifies environment-specific beans (e.g., `dev`, `test`, `prod`).
- **@PropertySource:** Loads properties from an external `.properties` file.
- **@EnableConfigurationProperties:** Binds external configuration properties to Java objects.

10. Persistence Annotations (JPA):

- `@Entity` → Marks a class as a database entity.
- `@Table` → Specifies the table name in the database.
- `@Id` → Defines the primary key.
- `@GeneratedValue` → Auto-generates primary key values.
- `@Column` → Maps class fields to database columns.
- `@Repository` → Indicates a data access layer (DAO).

11. Error Handling:

- **@ControllerAdvice:** Global exception handling across all controllers.
- **@ExceptionHandler:** Handles exceptions locally within a specific controller.

12. Testing Annotations:

- `@SpringBootTest` → Used for integration testing.
- `@MockBean` → Creates mock objects for testing.

13. Custom Annotations:

You can create custom annotations using `@interface`. Example:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface LogExecutionTime {}
```

⚙️ Spring Boot Configuration Overview

Spring Boot provides flexible configuration options to manage application settings. Here's a summary:

1. application.properties

- **Purpose:** Defines key-value pairs for application configuration (e.g., server port, database URL).
    - **Example:**

```
server.port=8081
spring.datasource.url=jdbc:mysql://localhost:3306/db
logging.level.org.springframework=DEBUG
```

2. application.yml (YAML)

- **Purpose:** A hierarchical format for structured configuration, making it more readable than `.properties`.
    - **Example:**

```
server:
  port: 8082
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/db
    username: user
    password: pass
```

3. bootstrap.properties / bootstrap.yml

- **Purpose:** Used in Spring Cloud applications for externalized configuration (e.g., connecting to Config Server).
    - **Example:**

```
spring.application.name=config-client
spring.cloud.config.uri=http://localhost:8888
```

4. application-{profile}.properties / .yml

- **Purpose:** Profile-specific configuration for different environments (dev, test, prod).
    - **Example:**

```
# application-dev.properties
server.port=8083
spring.datasource.username=dev_user
```

5. @Value Annotation

- **Purpose:** Injects configuration values directly into Java classes.
  - **Example:**

```
@Value("${server.port}")
private int port;
```

6. @ConfigurationProperties

- **Purpose:** Binds external configuration properties to Java objects for easy access.
  - **Example:**

```
@ConfigurationProperties(prefix = "app")
public class AppConfig {
    private String name;
    private String version;
}
```

7. Command-Line Arguments

- **Purpose:** Override configuration at runtime.
  - **Example:**

```
java -jar app.jar --server.port=9090
```

8. Environment Variables

- **Purpose:** Configure sensitive data (like passwords) in production environments.
  - **Example:**

```
export SPRING_DATASOURCE_PASSWORD=secret
```

9. XML Configuration (Rare in Spring Boot)

- **Purpose:** Legacy support; rarely used as Spring Boot favors annotations and properties.
  - **Example:**

```
<bean id="myService" class="com.example.MyService"/>
```

10. External Configuration Files

- **Purpose:** Load configuration from files outside the packaged JAR.
  - **Example:**

```
java -jar app.jar --spring.config.location=/path/to/config/
```

Common Port Numbers:

- HTTP: **80**
- HTTPS: **443**
- MySQL: **3306**

- PostgreSQL: **5432**
- MongoDB: **27017**
- Spring Boot (Default): **8080**

HTTP Status Codes:

- **200 OK:** Request successful
- **201 Created:** Resource successfully created
- **204 No Content:** Request successful, no content to return.
- **400 Bad Request:** Invalid request from client
- **401 Unauthorized:** Authentication required
- **403 Forbidden:** Access denied
- **404 Not Found:** Resource not found
- **500 Internal Server Error:** Server encountered an error
- **501 Not Implemented:** Server doesn't support the request method.
- **502 Bad Gateway:** Invalid response from the upstream server.
- **503 Service Unavailable:** Server temporarily unavailable (overloaded or down).
- **504 Gateway Timeout:** Server didn't get a timely response from another server.

Localhost URLs:

- **HTTP:** `http://localhost:8080`
- **HTTPS:** `https://localhost:8443`
- **Custom Port Example:** `http://localhost:3000` (React app), `http://localhost:4200` (Angular app)

Microservices - Key Concepts

Microservices:

An architectural style where applications are built as a collection of small, loosely coupled, independent services.

Why We Use Microservices:

- Scalability
- Flexibility in technology stack
- Faster development & deployment
- Improved fault isolation
- Easy maintenance

Microservice Architecture:

- **Description:** Each service handles a specific business function and communicates with other services via APIs (REST, gRPC, etc.).
- **Key Components:**
  - **Client:** Sends requests to the application.
  - **API Gateway:** Manages and routes requests to the correct microservice.
  - **Microservices:** Independent units for specific functionalities.
  - **Database:** Each service has its own database (Database per Service pattern).
  - **Service Discovery:** Helps locate other services dynamically.
  - **Load Balancer:** Distributes traffic evenly among services.
  - **Monitoring & Logging:** Tracks performance, errors, and activity logs.

Microservice Design Patterns:

- **API Gateway:** Acts as a single entry point for all client requests, handling routing, authentication, and rate limiting.
- **Database per Service:** Each microservice manages its own database, ensuring data isolation and flexibility.
- **Circuit Breaker:** Prevents cascading failures by stopping calls to a failing service after repeated failures.
- **Service Discovery:** Dynamically registers and discovers services, enabling scalability (e.g., Eureka).
- **Saga Pattern:** Manages distributed transactions across multiple services to maintain data consistency.

Basic Microservices Concepts:

- **Service Registry & Discovery (e.g., Eureka):** Maintains a registry of available services for dynamic discovery.
- **Load Balancing (e.g., Ribbon, Zuul):** Distributes incoming traffic across multiple service instances to optimize performance.
- **Communication:**
  - **REST:** HTTP-based synchronous communication.
  - **gRPC:** High-performance, contract-based communication.
  - **Messaging Queues (RabbitMQ, Kafka):** Asynchronous communication for event-driven architecture.
- **Fault Tolerance (Hystrix, Resilience4j):** Ensures system resilience by handling failures gracefully.
- **Containerization (Docker, Kubernetes):** Deploys microservices in isolated containers, simplifying scaling and management.
- **Logging & Monitoring (ELK Stack, Prometheus, Grafana):** Tracks logs, system metrics, and application health for performance monitoring.

REST API Concepts

What is a REST API?

REST (Representational State Transfer) API allows communication between client and server over HTTP by performing operations on resources using standard HTTP methods.

REST vs SOAP:

1. **REST:** Lightweight, flexible, uses HTTP, supports JSON, XML, etc.
2. **SOAP:** Heavyweight, uses XML, has strict standards, better for enterprise-level security.

RESTful Principles:

1. **Statelessness:** No client data is stored on the server between requests.
2. **Statefulness:** Server maintains the client state across multiple requests (less common in REST, more in traditional systems).
3. **Client-Server Architecture:** Separation of concerns between client and server.
4. **Cacheable:** Responses can be cached for better performance.
5. **Uniform Interface:** Standardized way to interact with resources.
6. **Layered System:** Multiple layers handle requests without the client knowing.

Real-Time Example:
Employee Management System:

1. `GET /api/employees` - Fetch employee list
2. `POST /api/employees` - Add new employee
3. `PUT /api/employees/1` - Update employee details
4. `DELETE /api/employees/1` - Delete employee record

Basic Syntax:

```
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    @GetMapping
    public List<Employee> getAllEmployees() {
    return employeeService.getAll();
    }

    @PostMapping
    public Employee createEmployee(@RequestBody Employee emp)
    {
    return employeeService.save(emp);
    }

    @PutMapping("/{id}")
    public Employee updateEmployee(@PathVariable Long id,
@RequestBody Employee emp) {
    return employeeService.update(id, emp);
    }

    @DeleteMapping("/{id}")
    public void deleteEmployee(@PathVariable Long id) {
    employeeService.delete(id);
    }
}
```

SQL Concepts -

Concept	Description	Example	Basic Syntax
Partitioning	Dividing a large table into smaller parts for better performance.	Partition sales data by year.	PARTITION BY year
LIMIT	Restricts the number of rows returned by a query.	Get top 5 employees.	SELECT * FROM employees LIMIT 5;
INNER JOIN	Returns matching rows from both tables.	Join employees with departments.	SELECT * FROM emp INNER JOIN dept ON emp.dept_id = dept.id;

Concept	Description	Example	Basic Syntax
LEFT JOIN	Returns all rows from the left table, even if there's no match.	List all employees with/without projects.	SELECT * FROM emp LEFT JOIN proj ON emp.id = proj.emp_id;
RIGHT JOIN	Returns all rows from the right table, even if there's no match.	List all projects with/without employees.	SELECT * FROM emp RIGHT JOIN proj ON emp.id = proj.emp_id;
FULL JOIN	Returns all rows when there's a match in one of the tables.	Combine employee and project data.	SELECT * FROM emp FULL JOIN proj ON emp.id = proj.emp_id;
UNION	Combines result sets of two queries, removing duplicates.	List cities from two tables.	SELECT city FROM A UNION SELECT city FROM B;
UNION ALL	Combines result sets, including duplicates.	List all cities from two tables.	SELECT city FROM A UNION ALL SELECT city FROM B;
GROUP BY	Groups rows sharing the same values for aggregate functions.	Count employees in each department.	SELECT dept, COUNT(*) FROM emp GROUP BY dept;
HAVING	Filters groups based on conditions (used with GROUP BY).	Departments with more than 5 employees.	SELECT dept, COUNT(*) FROM emp GROUP BY dept HAVING COUNT(*) > 5;
Subquery	Query within another query.	Employees with the highest salary.	SELECT * FROM emp WHERE salary = (SELECT MAX(salary) FROM emp);
Window Functions	Perform calculations across a set of table rows related to the current row.	Rank employees by salary.	SELECT name, salary, RANK() OVER (ORDER BY salary DESC) FROM emp;
Indexing	Speeds up data retrieval in large tables.	Index on employee ID.	CREATE INDEX idx_emp_id ON emp(id);

Concept	Description	Example	Basic Syntax
Normalization	Organizing data to reduce redundancy.	Splitting employee and department tables.	–
Denormalization	Combining tables to improve read performance.	Merging employee and department tables.	–
Constraints	Rules to maintain data integrity (e.g., PRIMARY KEY, UNIQUE).	Unique constraint on email.	ALTER TABLE emp ADD CONSTRAINT unique_email UNIQUE (email);
Transactions	Ensures a set of SQL operations are completed successfully together.	Bank transfer between accounts.	BEGIN; UPDATE A; UPDATE B; COMMIT;

MongoDB Concepts –

Concept	Description	Example	Basic Syntax
Database	Holds collections in MongoDB.	myDatabase	use myDatabase
Collection	Group of MongoDB documents (like tables).	employees	db.createCollection("employees")
Document	Basic unit of data (JSON-like structure).	{ name: "John", age: 30 }	db.employees.insertOne({ name: "John", age: 30 })
Insert	Add data to a collection.	Add employee data	db.employees.insertOne({ name: "Anna", dept: "HR" })
Find (Query)	Retrieve documents from a collection.	Find all employees	db.employees.find()
Filter	Apply	Employees	db.employees.find({ dept: "HR" })

Concept	Description	Example	Basic Syntax
(WHERE)	conditions to queries.	in HR	
Update	Modify existing documents.	Update age for John	db.employees.updateOne({ name: "John" }, { $set: { age: 35 } })
Delete	Remove documents from a collection.	Delete employee named Anna	db.employees.deleteOne({ name: "Anna" })
Aggregation	Process data with operations like sum, avg.	Total salaries per department	db.employees.aggregate([{ $group: { _id: "$dept", totalSalary: { $sum: "$salary" } } }])
Indexing	Improve query performance.	Index on name	db.employees.createIndex({ name: 1 })
Projection	Select specific fields to return in query.	Get only names	db.employees.find({}, { name: 1, _id: 0 })
Sorting	Sort documents by a field.	Sort by age	db.employees.find().sort({ age: 1 })
Limit	Restrict the number of documents returned.	Limit to 5 employees	db.employees.find().limit(5)
Skip	Skip specific number of documents in results.	Skip first 5 employees	db.employees.find().skip(5)
Joins (Lookup)	Perform joins between collections.	Join employees with departments	db.employees.aggregate([{ $lookup: { from: "departments", localField: "deptId", foreignField: "_id", as: "deptInfo" } }])

Concept	Description	Example	Basic Syntax
Partitioning	Distribute data across multiple servers.	Shard data for large collections	sh.enableSharding("myDatabase")
Transactions	Ensure atomicity of multiple operations.	Update salary and log together	session.startTransaction()
Schema Validation	Enforce data structure rules.	Validate age as a number	db.createCollection("employees", { validator: { $jsonSchema: { bsonType: "object", required: ["name", "age"], properties: { age: { bsonType: "int" } } } } })

DB2 Database Concepts -

Concept	Description	Example / Syntax
DB2	IBM's relational database management system for high-performance data.	-
Database	Organized collection of data.	CREATE DATABASE mydb;
Table	Stores data in rows and columns.	CREATE TABLE employees (id INT, name VARCHAR(50));
Schema	Logical grouping of database objects.	CREATE SCHEMA hr;
Primary Key	Unique identifier for each record.	PRIMARY KEY (id)
Foreign Key	Links two tables together.	FOREIGN KEY (dept_id) REFERENCES departments(id)
Insert Data	Adds new data to a table.	INSERT INTO employees VALUES (1, 'John Doe');
Select Data	Retrieves data from a table.	SELECT * FROM employees;
Update Data	Modifies existing data.	UPDATE employees SET name='Jane Doe' WHERE id=1;
Delete Data	Removes data from a table.	DELETE FROM employees WHERE id=1;
Joins	Combines rows from	SELECT * FROM A JOIN B ON

Concept	Description	Example / Syntax
	multiple tables.	A.id = B.id;
Partitioning	Divides data for performance.	CREATE TABLE emp PARTITION BY RANGE (emp_id);
Indexes	Speeds up data retrieval.	CREATE INDEX idx_name ON employees(name);
Views	Virtual table based on SELECT queries.	CREATE VIEW emp_view AS SELECT name FROM employees;
Triggers	Executes actions automatically on data changes.	CREATE TRIGGER trg AFTER INSERT ON employees FOR EACH ROW ...
Stored Procedure	Encapsulates business logic for reuse.	CREATE PROCEDURE proc_name() BEGIN ... END;
Functions	Returns a value after performing operations.	CREATE FUNCTION func_name() RETURNS INT BEGIN ... END;
Transactions	Ensures data integrity (ACID properties).	BEGIN; UPDATE ...; COMMIT;
Backup/Restore	Data protection mechanism.	BACKUP DATABASE mydb TO '/backup/path';
Constraints	Rules to enforce data integrity.	CHECK (salary > 0)
WITH Clause	Creates temporary result sets.	WITH temp AS (SELECT * FROM employees) SELECT * FROM temp;
LIMIT/OFFSET	Controls the number of records fetched.	SELECT * FROM employees FETCH FIRST 10 ROWS ONLY;
Locking	Controls concurrent access to data.	LOCK TABLE employees IN EXCLUSIVE MODE;

Git Basic Commands -

- `git init` - Initialize a new Git repository.
- `git clone <repo_url>` - Clone an existing repository.
- `git status` - Check the status of files in the working directory.
- `git add <file>` - Add specific file(s) to the staging area.
- `git add .` - Add all changes to the staging area.
- `git commit -m "message"` - Commit staged changes with a message.
- `git push` - Push commits to the remote repository.
- `git pull` - Fetch and merge changes from the remote repository.
- `git fetch` - Download changes from the remote repository without merging.

- `git branch` - List all branches.
- `git branch <branch_name>` - Create a new branch.
- `git checkout <branch_name>` - Switch to a specific branch.
- `git merge <branch_name>` - Merge changes from one branch to another.
- `git log` - View the commit history.
- `git diff` - Show changes between commits, branches, or working directory.
- `git reset --hard <commit_id>` - Reset to a specific commit (destructive).
- `git revert <commit_id>` - Revert changes from a specific commit (non-destructive).
- `git stash` - Temporarily save changes without committing.
- `git stash apply` - Reapply stashed changes.
- `git remote -v` - List remote repositories.

Angular Concepts

Components:
The core building blocks of Angular applications. They define the UI using HTML, CSS, and TypeScript. Each component consists of:

- **Template:** Defines the view (HTML).

- **Class:** Contains logic (TypeScript).
- **Styles:** Manages CSS for the component.
  Example: `@Component({ selector: 'app-header', templateUrl: './header.component.html' })`

Modules (`@NgModule`):
Organize an Angular app into cohesive blocks. Every Angular app has a root module (`AppModule`), and you can create feature modules for better modularity.
Example: `imports: [BrowserModule, AppRoutingModule]`

Data Binding:
Connects the UI (template) with the component's data.

- **Interpolation:** `{{ title }}`
- **Property Binding:** `[src]="imageUrl"`
- **Event Binding:** `(click)="handleClick()"`
- **Two-way Binding:** `[(ngModel)]="username"`

Directives:
Modify the DOM's structure or appearance.

- **Structural Directives:** Add or remove elements (`*ngIf`, `*ngFor`)
  Example: `<div *ngIf="isLoggedIn">Welcome!</div>`
- **Attribute Directives:** Change the behavior or style (`[ngClass]`, `[ngStyle]`)
  Example: `<p [ngClass]="{active: isActive}">Hello</p>`

Services & Dependency Injection (DI):
Services provide reusable logic (e.g., API calls, business logic). DI allows Angular to inject dependencies automatically.
Example:

```
@Injectable() export class DataService { getData() { return [1, 2, 3]; } }
constructor(private dataService: DataService) {}
```

Routing:
Enables navigation between different views/pages. Defined in `app-routing.module.ts`.
Example:

```
const routes: Routes = [ { path: 'home', component: HomeComponent }, { path: 'about', component: AboutComponent } ];
```

Forms (Reactive & Template-Driven):

- **Template-Driven Forms:** Simple, using `[(ngModel)]`.
  Example: `<input [(ngModel)]="name" required>`
- **Reactive Forms:** More robust, using `FormGroup`, `FormControl`.
  Example:

```
this.form = new FormGroup({ name: new FormControl('') });
```

HTTPClient:
Used for making API requests (GET, POST, PUT, DELETE).
Example:

```
this.http.get('https://api.example.com/data').subscribe(respons
e => console.log(response));
```

Lifecycle Hooks:
Control component behavior during its lifecycle.

- `ngOnInit`: Called once after the component is initialized (used for API calls, data fetching).
- `ngOnChanges`: Called when input properties change.
- `ngOnDestroy`: Clean up resources before the component is destroyed (unsubscribe from observables).
  Example:

```
ngOnInit() { console.log('Component initialized'); }
ngOnDestroy() { console.log('Component destroyed'); }
```

Pipes:
Transform data in templates.

- **Built-in Pipes:** `date`, `uppercase`, `currency`, etc.
  Example: `{{ today | date:'shortDate' }}`
- **Custom Pipes:** Create custom transformations.
  Example:

```
@Pipe({ name: 'capitalize' })
export class CapitalizePipe implements PipeTransform {
  transform(value: string): string { return
value.charAt(0).toUpperCase() + value.slice(1); }
}
```

Additional Angular Concepts

Observables & RxJS:
Used for handling asynchronous data streams (API calls, events).
Example:

```
this.http.get('api/data').subscribe(data => console.log(data));
```

Event Binding & Event Emitters:
Pass data from child to parent components using `@Output` and `EventEmitter`.
Example:

```
@Output() notify = new EventEmitter<string>();
this.notify.emit('Data from child');
```

Guards:
Protect routes from unauthorized access. Types include `CanActivate`, `CanDeactivate`.
Example:

```
canActivate(): boolean { return isLoggedIn; }
```

Interceptors:
Modify HTTP requests globally (add auth tokens, handle errors).
Example:

```
intercept(req: HttpRequest<any>, next: HttpHandler) {
  const cloned = req.clone({ headers:
req.headers.set('Authorization', 'Bearer token') });
  return next.handle(cloned);
}
```

Lazy Loading:
Load modules only when needed to improve performance.
Example:

```
{ path: 'admin', loadChildren: () =>
import('./admin/admin.module').then(m => m.AdminModule) }
```

Resolvers:
Fetch data before navigating to a route.
Example:

```
resolve() { return this.dataService.getData(); }
```

Standalone Components:
Angular 14+ feature allowing components without modules.
Example:

```
@Component({ standalone: true, selector: 'app-standalone',
template: '<p>Standalone!</p>' })
```

TrackBy Function (in `*ngFor`):
Optimizes performance by tracking items efficiently.
Example:

```
<div *ngFor="let item of items; trackBy:
trackById">{{ item.name }}</div>
```

Change Detection Strategy:
Controls how Angular detects and updates the UI.
Example:

```
@Component({ changeDetection: ChangeDetectionStrategy.OnPush })
```

Content Projection (`ng-content`):
Pass dynamic content into components.
Example:

```
<ng-content></ng-content>
```

Custom Directives:
Create your own directives to manipulate the DOM.
Example:

```
@Directive({ selector: '[appHighlight]' })
```