|  |
| --- |
|  |
|  |

|  |
| --- |
| **JOINS** |
| **✅ 1. INNER JOIN**   * **Returns** only matching rows from both tables.   **Example:**  SELECT s.name, c.courseName  FROM Student s  INNER JOIN Course c ON s.courseId = c.courseId;  🧾 *Returns students who are enrolled in a course.*  **✅ 2. LEFT JOIN (a.k.a. LEFT OUTER JOIN)**   * **Returns all rows** from the left table + **matching rows** from the right table. * If no match, NULLs for right-side columns.   **Example:**  SELECT s.name, c.courseName  FROM Student s  LEFT JOIN Course c ON s.courseId = c.courseId;  🧾 *Returns all students, even those not enrolled in any course.*  **✅ 3. RIGHT JOIN (a.k.a. RIGHT OUTER JOIN)**   * **Returns all rows** from the right table + **matching rows** from the left table.   **Example:**  SELECT s.name, c.courseName  FROM Student s  RIGHT JOIN Course c ON s.courseId = c.courseId;  🧾 *Returns all courses, even if no student is enrolled.*  **✅ 4. FULL OUTER JOIN**   * **Returns all rows** from both tables. * NULLs where there is no match.   **Example:**  SELECT s.name, c.courseName  FROM Student s  FULL OUTER JOIN Course c ON s.courseId = c.courseId;  🧾 *Returns all students and all courses — matching or not.*  **✅ 5. CROSS JOIN**   * Returns the **Cartesian product** — every row from table A with every row from table B.   **Example:**  SELECT s.name, c.courseName  FROM Student s  CROSS JOIN Course c;  🧾 *If 3 students and 2 courses → returns 6 rows (3×2).*  **✅ 6. SELF JOIN**   * Joins a table **to itself**, useful for hierarchical or comparison scenarios.   **Example:**  SELECT e1.name AS emp, e2.name AS manager  FROM Employee e1  JOIN Employee e2 ON e1.managerId = e2.empId;  🧾 *Shows each employee with their manager’s name.* |

|  |
| --- |
| **Design Patterns** |
| **✅ Creational Patterns**   * Singleton * Factory Method * Abstract Factory * Builder * Prototype   **🔧 Structural Patterns**   * Adapter * Flyweight * Facade   **🎯 Behavioral Patterns**   * Strategy * Observer * Decorator |

|  |
| --- |
| **Static Class** |
| In Java, only nested classes can be static. Top-level classes can't be static. A static nested class can be instantiated without an instance of the outer class, and it's used mainly for grouping classes logically and for better encapsulation. |

|  |
| --- |
| **Java Inner Classes** |
| 1. **Static Nested Class**    * A nested class declared with static keyword.    * It does NOT need an object of the outer class.    * Can only access static members of the outer class.   **Uses of Static Nested Classes in Java**   1. Logical Grouping of Classes:    * When a class is useful only within the context of another class, you can group it logically as a static nested class.    * Example: A Map class might have a nested MapEntry class. 2. Improved Encapsulation:    * You can hide the nested class from the outside world if it’s not needed elsewhere.    * Only the outer class and related code know about it. 3. Better Memory Efficiency:    * Since it doesn’t hold a reference to the outer class, it can be more lightweight than a non-static inner class. 4. Can Be Used Without Outer Class Object:    * Unlike non-static inner classes, static nested classes can be instantiated directly without creating an object of the outer class.   Outer.StaticNested obj = new Outer.StaticNested();   1. Useful in Utility or Helper Classes:    * Ideal for writing builder classes, constants, or helper methods that don’t need access to the outer class instance.   class Car {  private String engine;  private int wheels;  static class Builder {  private String engine;  private int wheels;  public Builder setEngine(String engine) {  this.engine = engine;  return this;  }  public Builder setWheels(int wheels) {  this.wheels = wheels;  return this;  }  public Car build() {  Car car = new Car();  car.engine = this.engine;  car.wheels = this.wheels;  return car;  }  }  }  Usage  Car car = new Car.Builder().setEngine("V8").setWheels(4).build();   1. **Non-Static Inner Class**  * A non-static inner class is a class declared inside another class without the static keyword. * It is tied to an instance of the outer class and can access all members (including private) of the outer class.   **Uses of Static Nested Classes in Java**   1. Tightly Coupled Logic:    * Use when the inner class logically needs access to instance-level data of the outer class. 2. Event Handlers in GUI Programming:    * Common in Swing/AWT when handling UI events (e.g., ActionListener). 3. Encapsulation:    * Helps in hiding the internal working of the outer class, useful in complex data structures. 4. Improved Code Readability:    * Keeping closely related classes together improves organization and maintainability.   class BankAccount {  private double balance = 1000;  class Transaction {  void deposit(double amount) {  balance += amount;  System.out.println("Deposited. New balance: " + balance);  }  }  }  Usage  BankAccount account = new BankAccount();  BankAccount.Transaction txn = account.new Transaction();  txn.deposit(500); |

|  |
| --- |
| **Optional** |
| Optional<T> is a container introduced in Java 8 to handle nullable values safely and avoid NullPointerException (NPE).   * Prevents NullPointerException by making nullability explicit. * Reduces excessive null checks. * Encourages functional programming (map, flatMap, etc.).  ****Creating Optional Instances**** Optional<String> opt1 = Optional.of("Hello"); // Throws NPE if null  Optional<String> opt2 = Optional.ofNullable(null); // Can hold null safely  Optional<String> opt3 = Optional.empty(); // Creates an empty Optional ****Retrieving Values**** String value1 = opt1.get(); // Use cautiously, throws if empty  String value2 = opt2.orElse("Default"); // Returns "Default" if empty  String value3 = opt2.orElseGet(() -> "Generated Default");  String value4 = opt2.orElseThrow(() -> new RuntimeException("Value not present")); ****Checking Presence of Value**** if(opt1.isPresent()) {  System.out.println(opt1.get());  }  opt1.ifPresent(System.out::println); // Prints "Hello" if present ****Transforming Values**** Optional<Integer> length = opt1.map(String::length); // Transforming value  Optional<String> result = Optional.of(Optional.of("Nested")).flatMap(opt -> opt); // Avoids nested Optional ****Best Practices**** ✅ Use Optional as a return type for methods that may return null. ❌ Avoid using Optional for **method parameters** and **class fields**. ****Real-World Example: Avoiding NullPointerException******Without Optional** public String getEmployeeName(Employee emp) {  if (emp != null) {  Address addr = emp.getAddress();  if (addr != null) {  return addr.getCity();  }  }  return "Unknown";  } **With Optional** public String getEmployeeName(Employee emp) {  return Optional.ofNullable(emp)  .map(Employee::getAddress)  .map(Address::getCity)  .orElse("Unknown");  } ****9. Output Scenarios****  | **Employee Object State** | **Output** | | --- | --- | | emp == null | "Unknown" | | emp.getAddress() == null | "Unknown" | | emp.getAddress().getCity() == null | "Unknown" | | emp.getAddress().getCity() == "New York" | "New York" |  ****10. Key Takeaways****  * Optional prevents NullPointerException and improves code readability. * Use orElse(), orElseGet(), and orElseThrow() for handling absent values. * Use map() and flatMap() for transformations. * Avoid using Optional in class fields and method parameters. |

|  |
| --- |
| **Function.identity()** |
| Function.identity() is a static method in the Function interface in Java. It returns a function that always returns its input argument. This can be useful when you need to pass a function as an argument but want the function to simply return the value it receives.  **Syntax:**  Function<T, T> identity()  **Usage:**   * It is commonly used in streams, map, groupingBy, or when dealing with transformations that don’t require any changes to the input. * It returns a function that takes an input and returns the same input without modification.   **Example:**  import java.util.function.Function;  public class IdentityFunctionExample {  public static void main(String[] args) {  Function<String, String> identityFunction = Function.identity();  String result = identityFunction.apply("Hello, World!");  System.out.println(result); // Output: Hello, World!  }  }  In this example, Function.identity() is used to create a function that simply returns the input string unchanged. |

|  |
| --- |
| **Static Imports** |
| A static import in Java allows you to import a class's static members (fields and methods) directly, so you can use them without prefixing them with the class name. Introduced in Java 5  **Syntax:**  import static package\_name.ClassName.staticMember;  **Example:**  **Without Static import** public class MathExample {  public static void main(String[] args) {  double result = Math.sqrt(25); // Using Math class explicitly  System.out.println("Square Root: " + result);  }  } **Without Static Import**import static java.lang.Math.sqrt; // Static importpublic class MathExample {public static void main(String[] args) {double result = sqrt(25); // No need for Math.sqrt()System.out.println("Square Root: " + result);}} **Example:Junit**  // Static import for assertions  import static org.junit.jupiter.api.Assertions.\*; import org.junit.jupiter.api.Test;class TestExample {@Testvoid testAddition() {int sum = 5 + 3;assertEquals(8, sum); // No need for Assertions.assertEquals()}} |

|  |
| --- |
| **Java Multithreading** |
| 🔒 Semaphore  * A semaphore controls access to a resource using a set number of permits. * Threads acquire permits before accessing the resource, and release them after use. * **Use case:** Limit concurrent database connections (say only 3 allowed at a time)   **Example:**  Semaphore semaphore = new Semaphore(2); // 2 permits  semaphore.acquire(); // get a permit  System.out.println("Accessing resource");  semaphore.release(); // release permit 🔒 ****Mutex (ReentrantLock)****  * A **Mutex (Mutual Exclusion)** allows only one thread to access a critical section. * Java uses ReentrantLock or synchronized as mutex. * **Use case:** Updating a shared counter or account balance.   **Example:**  Lock lock = new ReentrantLock();  lock.lock();  try {  System.out.println("Only one thread here");  } finally {  lock.unlock();  } 📝 ****Callable****  * Similar to Runnable, but:   + Can **return a value**   + Can **throw checked exceptions** * **Use case:** Background task returning a result like fetching data from an API.   **Example:**  Callable<Integer> task = () -> 10 + 20; 📦 ****Future****  * Represents the result of an asynchronous computation. * Can:   + get() result (blocks if not ready)   + cancel() the task   + isDone() to check if completed * **Use case:** Run a time-consuming task in the background and get result later.   **Example:**  ExecutorService executor = Executors.newSingleThreadExecutor();  Future<Integer> future = executor.submit(() -> 5 + 5);  System.out.println(future.get()); // wait and get result  executor.shutdown(); ⚙️ ****Executor****  * Interface for running Runnable tasks. * Decouples task submission from thread creation. * **Use case:** When you need a clean way to execute tasks without manually creating Threads.   **Example:**  Executor executor = Executors.newSingleThreadExecutor();  executor.execute(() -> System.out.println("Task running")); ⚙️ ****ExecutorService****  * Extends Executor with extra methods for managing threads:   + submit() for Runnable and Callable   + shutdown() to stop   + invokeAll() to run multiple tasks * **Use case:** When you want a pool of threads managed automatically.   **Example:**  ExecutorService service = Executors.newFixedThreadPool(2);  service.submit(() -> System.out.println("Task 1"));  service.shutdown(); 📌 ****Executors Utility Class****  * Factory class with static methods to create thread pools:   + newFixedThreadPool(int n)   + newCachedThreadPool()   + newSingleThreadExecutor()   + newScheduledThreadPool(int n) * **Use case:** Quickly create thread pools.   **Example:**  ExecutorService pool = Executors.newCachedThreadPool();  pool.execute(() -> System.out.println("Using cached pool"));  pool.shutdown(); 🗺️ ****ConcurrentHashMap****  * A thread-safe version of HashMap. * Uses **bucket-level locking** (lock striping). * No need to externally synchronize it. * **Use case:** Maintain online users in a multiplayer game or stock prices in trading.   **Example:**  ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();  map.put("A", 1);  System.out.println(map.get("A")); 📑 ****CopyOnWriteArrayList****  * Thread-safe ArrayList. * On every modification, it creates a **new copy** of the array. * Very fast for reads, expensive for writes. * **Use case:** Read-heavy configurations like active thread names, listeners.   **Example:**  CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();  list.add("Hello");  System.out.println(list); 📦 ****BlockingQueue****  * A thread-safe queue interface. * Methods:   + put() — waits if full   + take() — waits if empty * Implementations:   + ArrayBlockingQueue   + LinkedBlockingQueue   + PriorityBlockingQueue   + DelayQueue * **Use case:** Producer-consumer problems.   **Example:**  BlockingQueue<String> queue = new ArrayBlockingQueue<>(2);  queue.put("A");  System.out.println(queue.take()); |