|  |
| --- |
| **How and where do we use transactions in Hibernate, and what is the purpose of using them?** |
| In Hibernate, transactions are used to group a set of operations into a single unit of work that either **completes entirely** or **fails entirely**, ensuring **data consistency and integrity**. They are especially important when performing operations that change the database state, such as save, update, delete, or executing queries that modify data.  Hibernate uses the **Java Transaction API (JTA)** or **JDBC transactions**, depending on the environment. In a typical application using Hibernate with JDBC, we manage transactions using the org.hibernate.Transaction interface.  We usually begin and commit transactions like this:  Session session = sessionFactory.openSession();  Transaction tx = null;  try {  tx = session.beginTransaction();  // Perform DB operations  session.save(entity);  session.update(anotherEntity);    tx.commit(); // commits if all goes well  } catch (Exception e) {  if (tx != null) tx.rollback(); // rollback in case of error  e.printStackTrace();  } finally {  session.close(); }  **Where to use:** We use transactions in **DAO (Data Access Object)** layers or service layers where actual data persistence or retrieval happens.  **Purpose:**   1. **Atomicity**: Ensures all operations in the transaction succeed together or fail together. 2. **Consistency**: Maintains valid data by rolling back on failure. 3. **Isolation**: Keeps concurrent transactions from interfering with each other. 4. **Durability**: Once committed, changes are permanent even if a system failure occurs.   Using transactions also helps avoid issues like **partial updates**, **dirty reads**, or **lost updates**, making it a critical aspect of reliable enterprise application development. |

|  |
| --- |
| **What is Hibernate?** |
| Hibernate is an **Object-Relational Mapping (ORM) framework** for Java that simplifies database interactions. It allows developers to map Java objects to database tables, so we can work with databases using **Java objects instead of writing complex SQL queries**.  Hibernate handles all the **CRUD operations**, **transaction management**, **caching**, and even **lazy loading**. It abstracts away the JDBC boilerplate code, which improves productivity and reduces errors.  It also supports **HQL (Hibernate Query Language)**, which is object-oriented and more intuitive than SQL.  Hibernate makes applications more portable and maintainable by being **database-independent**—it can switch between databases without changing much code, thanks to its dialect system.  **In short**, Hibernate helps us **save, update, delete, and retrieve** Java objects directly to/from a relational database in an efficient and consistent way. |

|  |
| --- |
| **How does Spring Boot use Hibernate?** |
| Spring Boot uses Hibernate as the default JPA (Java Persistence API) provider to handle ORM (Object-Relational Mapping). When we add the spring-boot-starter-data-jpa dependency, Spring Boot automatically configures Hibernate behind the scenes to connect with the database and map Java entities to database tables.  Hibernate gets integrated into Spring Boot via Spring Data JPA, which simplifies repository access. We just define entity classes and interfaces that extend JpaRepository or CrudRepository, and Spring Boot—with Hibernate underneath—takes care of the rest.  Spring Boot also reads configuration properties from application.properties or application.yml to set up the Hibernate connection, dialect, DDL options (update, create, etc.), and connection pool.  Example: spring.datasource.url=jdbc:mysql://localhost:3306/mydb spring.datasource.username=root  spring.datasource.password=pass  spring.jpa.hibernate.ddl-auto=update  spring.jpa.show-sql=true  So, Hibernate handles the actual ORM logic, while Spring Boot provides the auto-configuration and abstraction to make it easier and faster to develop with. |

|  |
| --- |
| **spring.jpa.hibernate.ddl-auto=update** |
| The spring.jpa.hibernate.ddl-auto property controls how Hibernate handles schema generation and management when your Spring Boot application starts.   | **Option** | **Description** | | --- | --- | | none | **No action**: Hibernate won’t manage the schema at all. Use this in production when schema is already in place. | | validate | Hibernate **validates** the schema against entity mappings. It throws an error if there’s a mismatch (but doesn’t modify the schema). | | update | Hibernate **updates** the schema to match entity definitions — adds new columns/tables but doesn’t drop existing ones. Great for development. | | create | Drops the existing schema (if any) and **creates a new schema** from scratch every time the app starts. | | create-drop | Same as create, but also **drops the schema** when the session factory is closed (e.g., when the app shuts down). | | none | Hibernate doesn’t perform any DDL actions. You manage schema manually. | |

|  |
| --- |
| **Explain Singleton and Factory Design Patterns with pseudo code.** |
| **✅ 1. Singleton Pattern**  **Answer:**  The Singleton pattern ensures that **only one instance** of a class is created during the application's lifetime and provides a **global access point** to that instance.  **Use case:** Logging, configuration, DB connection managers.  **Pseudo-code:**  class Singleton {  private static Singleton instance = null;  private Singleton() {  }  public static Singleton getInstance() {  if (instance == null) {  instance = new Singleton(); // lazy initialization  }  return instance;  }  }  **✅ 2. Factory Pattern**  **Answer:** The Factory pattern provides a way to **create objects without specifying the exact class** of the object that will be created. It delegates the instantiation logic to a method (the factory method).  **Use case:** When you have a common interface or superclass but need different implementations based on input.  **Pseudo-code:**  interface Shape {  void draw();  }  class Circle implements Shape {  public void draw() {  print("Drawing Circle");  }  }  class Square implements Shape {  public void draw() {  print("Drawing Square");  }  }  class ShapeFactory {  public Shape getShape(String type) {  if (type == "CIRCLE") return new Circle();  else if (type == "SQUARE") return new Square();  else return null;  }  }  **Usage:**  ShapeFactory factory = new ShapeFactory();  Shape shape1 = factory.getShape("CIRCLE");  shape1.draw(); // Output: Drawing Circle |

|  |
| --- |
| **What is Spring Boot Actuator and what are its different methods/endpoints?** |
| Spring Boot Actuator is a powerful feature that provides **production-ready monitoring and management** of a Spring Boot application. It exposes various **REST endpoints** that give insights into the application's health, metrics, environment, thread dumps, and more—without writing extra code.  To enable it, we add this dependency:  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  By default, some endpoints are enabled and secured. We can configure them in application.properties.  **🔧 Common Actuator Endpoints (Methods):**   | **Endpoint** | **Description** | | --- | --- | | /actuator/health | Shows application health status (UP/DOWN). | | /actuator/info | Displays custom app info (can be set in properties). | | /actuator/metrics | Exposes metrics like memory, CPU, GC, etc. | | /actuator/env | Shows current environment properties. | | /actuator/beans | Lists all Spring beans in the context. | | /actuator/mappings | Shows all request-to-handler mappings. | | /actuator/heapdump | Provides a heap dump for memory analysis. | | /actuator/threaddump | Displays thread dump of running threads. | | /actuator/loggers | Allows you to view or change log levels at runtime. |   We can enable or expose these using:  management.endpoints.web.exposure.include=health,info,metrics,env  **In summary**, Spring Boot Actuator helps in **monitoring, debugging, and managing** applications—especially useful in microservices and production environments. |

|  |
| --- |
| **What are the key Java 8 features like Streams, Lambda expressions, and Functional Interfaces? Where are they used?** |
| Java 8 introduced powerful features that made code more **concise, readable, and functional**, especially when working with collections and functional-style programming.  **✅ 1. Lambda Expressions**  Lambda expressions provide a way to write **anonymous functions** (i.e., methods without a name). They reduce boilerplate when writing implementations for interfaces with a **single abstract method**.  **Syntax:**  (parameter) -> expression  **Example:**  List<String> names = Arrays.asList("A", "B", "C");  names.forEach(name -> System.out.println(name)); // concise loop  **Where to use:**   * Inside forEach, map, filter, sort, or any functional interface implementation (like Runnable, Comparator).   **✅ 2. Streams API**  Streams allow you to **process collections in a functional style**—filtering, mapping, and reducing data with minimal code and high performance.  **Example:**  List<String> names = Arrays.asList("John", "Jane", "Jack");  List<String> filtered = names.stream()  .filter(n -> n.startsWith("J"))  .collect(Collectors.toList());  **Where to use:**   * Whenever you need to process collections (e.g., filter, sort, transform data) with better readability and performance.   **✅ 3. Functional Interfaces**  A **functional interface** is any interface with **a single abstract method (SAM)**. They can be implemented using lambda expressions.  **Common types:**   |  |  |  |  | | --- | --- | --- | --- | | **Interface** | **Method** | **Purpose** | **Example** | | Function<T,R> | R apply(T t) | Convert from T to R | Convert String to Integer | | Predicate<T> | boolean test(T t) | Check a condition | Check if number > 10 | | Consumer<T> | void accept(T t) | Consume input, no return | Print a value | | Supplier<T> | T get() | Supply a value without input | Generate random number |   **Example:**  Predicate<String> isEmpty = str -> str.isEmpty(); System.out.println(isEmpty.test("")); // true |

|  |
| --- |
| **What is the difference between Abstraction and Interface in Java? Can objects be created from them?** |
| **Abstraction** and **Interface** both support abstraction in Java, but they are used differently and serve different design needs.  **🔍 Key Differences:**   |  |  |  | | --- | --- | --- | | **Feature** | **Abstract Class** | **Interface** | | Methods | Can have both **abstract** and **non-abstract** methods | Only **abstract methods** (Java 7); from Java 8, can also have **default** and **static** methods | | Variables | Can have instance variables | Only constants (public static final) | | Constructors | ✅ Yes | ❌ No (interfaces can’t have constructors) | | Inheritance | Supports **single** inheritance | Supports **multiple inheritance** | | Object Creation | ❌ Cannot instantiate directly | ❌ Cannot instantiate directly |   **✅ Real-Time Use Case:**   * **Abstract Class:** Use when classes share common behavior and state. Example: In a **Banking App**, you might have an abstract class Account with common logic (like deposit, withdraw) shared by subclasses SavingsAccount and CurrentAccount. * **Interface:** Use to define a contract or capability across unrelated classes. Example: Printable interface can be implemented by both Invoice and Report, even though they are unrelated types.   **🧪 Object Creation?**   * **No**, you **cannot directly create objects** from an abstract class or interface. * But you can create objects from **classes that extend an abstract class** or **implement an interface**.   abstract class Vehicle {  abstract void start();  }  class Car extends Vehicle {  void start() { System.out.println("Car starting..."); }  }  Vehicle v = new Car(); // Allowed: Upcasting  java  CopyEdit  interface Engine {  void run();  }  class DieselEngine implements Engine {  public void run() { System.out.println("Running diesel engine"); }  }  Engine e = new DieselEngine(); // Allowed  **In short**, use an **abstract class** when you need base functionality with shared code and state; use an **interface** when you want to enforce a contract across unrelated classes.  Would you like a quick diagram comparing both visually? |

|  |
| --- |
| **What is JIT? How JIT is different from normal compiler?** |
| **JIT stands for Just-In-Time compiler.** It is a component of the Java Virtual Machine (JVM) that improves the performance of Java applications by converting bytecode into native machine code **at runtime**. This allows the program to run faster because the JVM doesn't have to interpret the bytecode repeatedly.  **The main difference between JIT and a normal compiler** is the timing of compilation:   * A **normal compiler** like javac compiles Java source code into bytecode **before runtime**. * The **JIT compiler** works **at runtime**, converting frequently used bytecode into optimized machine code, based on how the program is actually running.   This means JIT can perform **runtime optimizations**, making execution faster, especially for code that's used repeatedly.  **🔁 Example:**  For example, in Java, the javac compiler compiles .java files into .class files (bytecode), which is platform-independent. At runtime, the JVM uses JIT to compile parts of this bytecode into native code for better speed. |

|  |
| --- |
| **Serialization** |
| Serialization is the process of converting an object into a byte stream, so it can be saved to a file or transferred over a network. We use the Serializable interface in Java for this. It’s useful for caching, session storage, etc. |
| **Reflection** |
| Reflection is an API in Java used to inspect and manipulate classes, methods, and fields at runtime. It’s powerful but should be used carefully because it can **break encapsulation** and even **bypass Singleton pattern**. |
| **Synchronization** |
| Synchronization is used to control access to shared resources in multithreading environments. It prevents thread interference and ensures **thread safety**, especially when dealing with shared data  synchronized (this) {  // thread-safe block } |
| **Cloning** |
| Cloning allows us to create a **copy of an object** using the clone() method from the Object class. To use cloning, a class should implement the Cloneable interface. It performs a **shallow copy** by default. |

|  |
| --- |
| **Breaking Singleton Class – Code Examples** |
| **Reflection:**[Reflection](https://www.geeksforgeeks.org/reflection-in-java/) can be caused to destroy singleton property of the singleton class.  **Overcome reflection issue:** To overcome issues raised by reflection, [enums](https://www.geeksforgeeks.org/enum-in-java/) are used because java ensures internally that the enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization.  As the constructor for an enum is **package-private** or **private access**, It automatically creates the constants that are defined at the beginning of the enum body. You cannot invoke an enum constructor yourself, so it is not possible for Reflection to utilize it. Hence, reflection can’t break singleton property in the case of enums.  **Serialization:-** [Serialization](https://www.geeksforgeeks.org/serialization-in-java/) can also cause breakage of singleton property of singleton classes. Serialization is used to convert an object of a byte stream and save in a file or send it over a network. Suppose you serialize an object of a singleton class. Then if you de-serialize that object it will create a new instance and hence break the singleton pattern.  **Overcome serialization issue:** To overcome this issue, we have to implement the method readResolve() method. readResolve() is a **special method** used during **deserialization** to **replace the deserialized object** with another object, typically the existing singleton instance.  protected Object readResolve() {  return getInstance();  }  **Cloning:**[Cloning](https://www.geeksforgeeks.org/clone-method-in-java-2/) is the concept to create duplicate objects. Using clone we can create copy of object. Suppose we create clone of a singleton object, then it will create a copy that is there are two instances of a singleton class, hence the class is no more singleton.  **Overcome Cloning issue:**To overcome this issue, override clone() method and throw an exception from clone method that is CloneNotSupportedException. Now, whenever user will try to create clone of singleton object, it will throw an exception and hence our class remains singleton.  import java.io.\*;  // Step 1: Implement Serializable and Cloneable to handle edge cases  public class Singleton implements Serializable, Cloneable {  // Step 2: Eagerly create instance for simplicity and thread-safety  private static final Singleton INSTANCE = new Singleton();  // Step 3: Flag to detect Reflection attack  private static boolean instanceCreated = false;  // Step 4: Private constructor to prevent external instantiation  private Singleton() {  // Prevent Reflection: Throw exception if instance already exists  if (instanceCreated) {  throw new RuntimeException("Use getInstance() method to create the Singleton");  }  instanceCreated = true;  }  // Step 5: Public method to provide global access point with synchronization  public static synchronized Singleton getInstance() {  return INSTANCE;  }  // Step 6: Prevent Serialization from creating a new instance  protected Object readResolve() {  return INSTANCE;  }  // Step 7: Prevent Cloning from creating a new instance  @Override  protected Object clone() throws CloneNotSupportedException {  throw new CloneNotSupportedException("Cloning a Singleton is not allowed");  }  // Utility method for demo  public void showMessage() {  System.out.println("Hello from Singleton!");  }  // MAIN METHOD to test all protections  public static void main(String[] args) throws Exception {  Singleton instance1 = Singleton.getInstance();  // 1. Serialization Test  ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("singleton.obj"));  out.writeObject(instance1);  out.close();  ObjectInputStream in = new ObjectInputStream(new FileInputStream("singleton.obj"));  Singleton instance2 = (Singleton) in.readObject();  in.close();  System.out.println("Serialization check: " + (instance1 == instance2)); // true  // 2. Reflection Test  try {  java.lang.reflect.Constructor<Singleton> constructor = Singleton.class.getDeclaredConstructor();  constructor.setAccessible(true);  Singleton instance3 = constructor.newInstance(); // should throw exception  } catch (Exception e) {  System.out.println("Reflection check passed: " + e.getMessage());  }  // 3. Cloning Test  try {  Singleton instance4 = (Singleton) instance1.clone(); // should throw exception  } catch (Exception e) {  System.out.println("Cloning check passed: " + e.getMessage());  }  }  } |

|  |
| --- |
| **Singleton implementation using lazy initialization with double-checked locking and protection against: ✅ Reflection✅ Serialization✅ Cloning✅ Thread-safety with synchronization** |
| import java.io.\*;  // Singleton class with all protections  public class Singleton implements Serializable, Cloneable {  // Volatile ensures visibility and ordering of writes across threads  private static volatile Singleton instance = null;  // Flag to prevent reflection-based attacks  private static boolean instanceCreated = false;  // Private constructor to prevent instantiation  private Singleton() {  if (instanceCreated) {  throw new RuntimeException("Use getInstance() method");  }  instanceCreated = true;  }  // Double-checked locking for thread-safe lazy initialization  public static Singleton getInstance() {  if (instance == null) {  synchronized (Singleton.class) {  if (instance == null) {  instance = new Singleton();  }  }  }  return instance;  }  // Prevent creating new instance during deserialization  protected Object readResolve() {  return getInstance();  }  // Prevent object cloning  @Override  protected Object clone() throws CloneNotSupportedException {  throw new CloneNotSupportedException("Cloning a Singleton is not allowed");  }  // Sample method  public void showMessage() {  System.out.println("Hello from lazy Singleton!");  }  // Main method for testing  public static void main(String[] args) throws Exception {  Singleton s1 = Singleton.getInstance();  // 1. Serialization Test  ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("singleton.obj"));  out.writeObject(s1);  out.close();  ObjectInputStream in = new ObjectInputStream(new FileInputStream("singleton.obj"));  Singleton s2 = (Singleton) in.readObject();  in.close();  System.out.println("Serialization check: " + (s1 == s2)); // true  // 2. Reflection Test  try {  java.lang.reflect.Constructor<Singleton> constructor = Singleton.class.getDeclaredConstructor();  constructor.setAccessible(true);  Singleton s3 = constructor.newInstance(); // should throw exception  } catch (Exception e) {  System.out.println("Reflection check passed: " + e.getMessage());  }  // 3. Cloning Test  try {  Singleton s4 = (Singleton) s1.clone(); // should throw exception  } catch (Exception e) {  System.out.println("Cloning check passed: " + e.getMessage());  }  }  } |

|  |
| --- |
| **What is volatile in Java?** |
| **volatile** is a **keyword in Java** used to indicate that a variable's value may be modified by multiple threads. When a variable is declared volatile, it ensures **visibility** of changes to that variable across all threads.  In simple terms, when one thread updates a volatile variable, the new value is immediately visible to all other threads. This prevents threads from caching the variable locally and using stale values.  private static volatile Singleton instance;  Java threads typically cache variables for performance.  Without volatile, one thread may not see changes made by another, because it's reading a stale cached value.  Declaring a variable volatile disables thread-local caching for that variable.  All reads and writes go directly to main memory (RAM), not the CPU cache. |

|  |
| --- |
| **What is an atomic variable in Java?** |
| In Java, atomic variables are part of the java.util.concurrent.atomic package and are used to perform thread-safe operations on single variables without using synchronized blocks.  They provide atomicity, meaning the operation is completed as a single, indivisible step, even in a multithreaded environment.  AtomicInteger count = new AtomicInteger(0);  count.incrementAndGet(); // Atomically increments by 1  Common Classes in java.util.concurrent.atomic:   * AtomicInteger * AtomicLong * AtomicBoolean * AtomicReference<T> |

|  |
| --- |
| **Why use atomic variables over volatile?** |
| While both volatile and atomic variables deal with multithreading and shared memory, they serve different purposes:  volatile int count = 0;  public void increment() {  count++; // NOT atomic: read-modify-write can be interrupted  }  AtomicInteger count = new AtomicInteger(0);  public void increment() {  count.incrementAndGet(); // Atomic and thread-safe. This avoids race conditions without locks.  }  Use \*\*volatile\*\* when you're just using a flag or checking a single read/write value (like stopThread = true).  Use \*\*Atomic variables\*\* when you're doing compound operations (like incrementing, updating, or compare-and-set). |

|  |
| --- |
| **What is a synchronized keyword and synchronization block in Java?** |
| In Java, the \*\*synchronized\*\* keyword is used to control access to a block of code or method by multiple threads, ensuring that only one thread can execute it at a time.  It is used to prevent race conditions and ensure thread safety when multiple threads are accessing shared resources.  When you declare a method as synchronized, it locks the **entire method** for the object instance or the class (if the method is static). This ensures that only one thread can execute this method on the same object at a time.  A synchronized block allows you to **lock a specific section of code**, giving you more control. You can synchronize a smaller part of your code rather than the whole method, and you can specify which object the thread should lock on.  public void increment() {  synchronized(this) {  count++;  }  } |

|  |
| --- |
| **What is Covariance in Java?** |
| Covariance refers to the ability of a subtype to be used in place of its supertype, typically involving types that maintain the relationship between objects in inheritance hierarchies. In Java, it primarily applies to generics and arrays.  **Covariant Arrays:**  In Java, arrays are covariant. This means that an array of a subclass can be assigned to an array of its superclass.  class Animal {}  class Dog extends Animal {}  public class CovariantExample {  public static void main(String[] args) {  Dog[] dogs = new Dog[10];  Animal[] animals = dogs; // Covariant, allowed  animals[0] = new Dog(); // Valid  animals[1] = new Animal(); // Throws ArrayStoreException at runtime!  }  }  Java generics are not covariant by default because generics are invariant. However, you can achieve covariance using wildcards in generics.  class Animal {}  class Dog extends Animal {}  class Cat extends Animal {}  public class CovariantGenericsExample {  public static void main(String[] args) {  List<? extends Animal> animals; // Covariant list  List<Dog> dogs = new ArrayList<>();  animals = dogs; // Allowed: List<Dog> is a subtype of List<? extends Animal>  List<Cat> cats = new ArrayList<>();  animals = cats; // Also allowed: List<Cat> is a subtype of List<? extends Animal>  }  }  Covariant arrays: An array of a subclass can be assigned to a variable of its superclass type, but inserting objects of the superclass into a subclass array may result in ArrayStoreException.  Covariant generics: Achieved using wildcards (? extends T), allowing you to use a subclass of a generic type as a supertype (e.g., List<? extends Animal>). |

|  |
| --- |
| **How can you sort a list of objects by a specific field (e.g., itemCategory)? Explain with both Comparable and Comparator approaches. Which one should you choose and when?** |
| In Java, you can sort objects by a specific field using either:   1. The Comparable interface (natural ordering) 2. The Comparator interface (custom ordering)   1. Using Comparable (Natural Ordering)  Use this when your class has a default sort rule (like sorting products by name or price).  class Item implements Comparable<Item> {  String name;  String itemCategory;  public Item(String name, String itemCategory) {  this.name = name;  this.itemCategory = itemCategory;  }  @Override  public int compareTo(Item other) {  return this.itemCategory.compareTo(other.itemCategory);  }  }  List<Item> items = new ArrayList<>();  items.add(new Item("Pen", "Stationery"));  items.add(new Item("Apple", "Fruit"));  Collections.sort(items); // uses compareTo() defined in Comparable  2. Using Comparator (Custom Ordering)  Use this when:   * You don’t want to modify the class. * You want to sort the same object in different ways (by category, name, price, etc.)   class Item {  String name;  String itemCategory;  public Item(String name, String itemCategory) {  this.name = name;  this.itemCategory = itemCategory;  }  }  class CategoryComparator implements Comparator<Item> {  @Override  public int compare(Item a, Item b) {  return a.itemCategory.compareTo(b.itemCategory);  }  }  ==========================  List<Item> items = new ArrayList<>();  items.add(new Item("Pen", "Stationery"));  items.add(new Item("Apple", "Fruit"));  Collections.sort(items, new CategoryComparator());   |  |  |  | | --- | --- | --- | | **Criteria** | **Comparable** | **Comparator** | | Defined inside the class? | Yes (compareTo method) | No (external to the class) | | One sorting strategy | Yes (natural ordering) | Multiple strategies supported | | Modifies original class? | Yes | No | | Use case | Default or single sorting logic | Multiple sorting needs or 3rd-party classes |  * Use Comparable when you want to provide a default natural ordering inside the class itself. * Use Comparator when you want flexibility to sort by different fields or avoid modifying the class. |

|  |
| --- |
| **What is Synchronization in Java? What Happens When You Use a synchronized Method?** |
| Synchronization in Java is a way to control access to shared resources by multiple threads. It ensures that only one thread can access the critical section (a block of code that modifies shared data) at a time, helping to prevent race conditions and ensure data consistency in a multithreaded environment.  When a method is declared with the synchronized keyword, the thread must acquire the intrinsic lock (monitor) for the object before it can execute the method.  There are two types of synchronized methods:  🔹 1. Instance Synchronized Method  public synchronized void printData() {  // critical section  }   * Acquires the lock on the current object (this). * Only one thread can call any other synchronized method on the same object at a time.   🔹 2. Static Synchronized Method  public static synchronized void log() {  // static critical section  }   * Acquires the lock on the class object (ClassName.class). * Useful when working with static data shared across all instances.  |  |  | | --- | --- | | **Feature** | **Behavior** | | Mutual exclusion | Only one thread at a time can enter the synchronized method/block | | Visibility guarantee | Changes made by one thread are visible to others after releasing the lock | | Performance impact | Slight overhead due to locking and context switching | | Deadlock risk (if misused) | Can occur if multiple threads wait on each other’s locks |   **Synchronization in Java** makes sure that only **one thread** can access (mutual exclusion) a **shared piece of code** at a time (shared resources). This prevents problems when multiple threads try to change the same data.  A **synchronized method** puts a **lock** on the object, so other threads must wait their turn before they can run that method. |

|  |
| --- |
| **Can we execute any code before the main() method in Java? How?** |
| Yes, in Java, you can execute code before the main() method using a static block. Static blocks are executed when the class is loaded into memory by the JVM, even before the main() method is called.  public class Demo {  static {  System.out.println("Static block executed before main()");  }  public static void main(String[] args) {  System.out.println("Inside main method");  }  }  Static block executed before main()  Inside main method  For initializing static variables  For logging or loading config files  For setup tasks before main starts |

|  |
| --- |
| **What is an Instance Block in Java and What Is It Used For?** |
| An instance block (also called an instance initializer block) in Java is a block of code that is executed every time an object is created, before the constructor runs.  {  // instance block  System.out.println("Instance block called");  }  public class Demo {  {  System.out.println("Instance block executed");  }  public Demo() {  System.out.println("Constructor executed");  }  public static void main(String[] args) {  Demo d1 = new Demo();  Demo d2 = new Demo();  }  }  Instance block executed  Constructor executed  Instance block executed  Constructor executed  Common initialization code for all constructors  Code that should run every time an object is created  When you don’t want to repeat logic in multiple constructors |

|  |
| --- |
| **Count the frequency of each character from a list of strings using Java Streams. The output should display characters and their counts in the order they appear.** |
| import java.util.\*;  import java.util.stream.\*;  public class CharacterFrequency {  public static void main(String[] args) {  List<String> list1 = List.of("apple", "orange");  String result3 = list1.stream()  .flatMapToInt(String::chars)  .mapToObj(c -> (char) c)  .collect(Collectors.groupingBy(  c -> c,  LinkedHashMap::new,  Collectors.counting()  ))  .entrySet().stream()  .map(e -> e.getKey() + "-" + e.getValue())  .collect(Collectors.joining(", "));  System.out.println(result3);  }  }  list1.stream()  Creates a stream of the list elements:  Stream<String> -> "apple", "orange"  .flatMapToInt(String::chars)  Each string is converted to its character code points (int values). So:  "apple" → 'a', 'p', 'p', 'l', 'e'  "orange" → 'o', 'r', 'a', 'n', 'g', 'e'  Resulting IntStream (all characters in order, flattened):  ['a', 'p', 'p', 'l', 'e', 'o', 'r', 'a', 'n', 'g', 'e']  .mapToObj(c -> (char) c)  Converts each int back to a char object, resulting in:  ['a', 'p', 'p', 'l', 'e', 'o', 'r', 'a', 'n', 'g', 'e']  .collect(Collectors.groupingBy(c -> c, LinkedHashMap::new, Collectors.counting()))  Groups by character and counts occurrences,  using a LinkedHashMap to maintain insertion order.  Intermediate result:  {  'a' -> 2,  'p' -> 2,  'l' -> 1,  'e' -> 2,  'o' -> 1,  'r' -> 1,  'n' -> 1,  'g' -> 1  }  .entrySet().stream()  .map(e -> e.getKey() + "-" + e.getValue())  Converts each entry to a string of the form: char-count  Example: 'a'=2 becomes "a-2"  Intermediate list:  ["a-2", "p-2", "l-1", "e-2", "o-1", "r-1", "n-1", "g-1"]  .collect(Collectors.joining(", "))  Joins the strings with comma + space:  "a-2, p-2, l-1, e-2, o-1, r-1, n-1, g-1"  Final Output  a-2, p-2, l-1, e-2, o-1, r-1, n-1, g-1 |

|  |
| --- |
| **Illustrate flatMap with a real-world example. Also show the output with and without using flatMap.** |
| flatMap is a method in Java Streams used to transform each element of a stream into a new stream and then flatten all those streams into a single continuous stream.  You have a list of students, and each student has a list of hobbies. You want a single list of all hobbies.  class Student {  String name;  List<String> hobbies;  Student(String name, List<String> hobbies) {  this.name = name;  this.hobbies = hobbies;  }  List<String> getHobbies() {  return hobbies;  }  }  Using flatMap  public class FlatMapExample {  public static void main(String[] args) {  List<Student> students = List.of(  new Student("Alice", List.of("Reading", "Swimming")),  new Student("Bob", List.of("Cycling", "Gaming"))  );  List<String> allHobbies = students.stream()  .flatMap(student -> student.getHobbies().stream())  .collect(Collectors.toList());  System.out.println("Using flatMap: " + allHobbies);  }  }  Using flatMap: [Reading, Swimming, Cycling, Gaming]  Without flatMap (using map)  public class MapExample {  public static void main(String[] args) {  List<Student> students = List.of(  new Student("Alice", List.of("Reading", "Swimming")),  new Student("Bob", List.of("Cycling", "Gaming"))  );  List<List<String>> hobbyLists = students.stream()  .map(Student::getHobbies)  .collect(Collectors.toList());  System.out.println("Using map: " + hobbyLists);  }  }  Using map: [[Reading, Swimming], [Cycling, Gaming]]  **flatMap() Output**  List of Lists  [[Reading, Swimming], [Cycling, Gaming]]   |  | | --- | |  |  |  | | --- | | When you want to keep nested structure |   **map() Output**  One flat List  [Reading, Swimming, Cycling, Gaming]  When you want to flatten results |

|  |
| --- |
|  |
|  |