

# Java Interview - Gemini

## ☕ Java Core

---

What is the difference between JVM, JDK and JRE?

- **JVM (Java Virtual Machine):** The engine 🌀. It's an abstract machine that provides the runtime environment for Java bytecode to be executed.
- **JRE (Java Runtime Environment):** The pit crew 🛠️. It includes the JVM and the core class libraries needed to run a Java application. It doesn't have development tools.
- **JDK (Java Development Kit):** The full garage 🚗. It includes everything in the JRE, plus development tools like the compiler (`javac`), debugger, and other utilities. You need the JDK to **develop** Java applications.

**Real-world example:** A software developer at a company like **JetBrains** uses the **JDK** to write and compile code for an IDE like IntelliJ. The end-user of a banking application only needs the **JRE** to run the `.jar` file on their machine. The `.jar` file's bytecode is then executed by the **JVM**.

---

What is the default value of not-initialised properties of the class `boolean b` and `Integer i`?

- For a class property `boolean b`, the default value is `false`.
  - For a class property `Integer i` (which is an object), the default value is `null`.
- 

What is the default value of not-initialised variables inside local method `boolean b` and `Integer i`?

Local variables have **no default value**. The compiler will raise a compile-time error if you attempt to use a local variable before it has been explicitly initialized.

---

How to initialize `final` variables?

A `final` variable can only be assigned a value once. This can be done in three ways:

1. **At declaration:** `final int MAX_ATTEMPTS = 3;`
2. **Inside an instance initializer block:**  
Java

```
final String serverId;
{
    serverId = "prod-api-01";
}
```

1. **Inside the constructor:** This is the most common approach for setting a `final` value based on a constructor parameter.

Java

```
final String userId; public User(String userId) {
    this.userId = userId;
}
```

---

How are strings passed to a method - by reference or by value? What about primitive booleans?

Java is **strictly pass-by-value** ➡.

- When an object like a `String` is passed, a **copy of the reference** to that object is passed. The method can't change what the original reference points to, but it can access the same object.
- When a primitive like a `boolean` is passed, a **copy of the value** is passed. Any changes made to the parameter inside the method won't affect the original variable.

---

Where are booleans and Strings stored in the JVM?

- Primitive `boolean` variables are stored on the **stack** (for method-local variables) or the **heap** (as part of a class instance).
- `String` objects are stored on the **heap**. `String` literals are stored in a special area of the heap called the **String Pool** for optimization and memory efficiency.

---

What is the difference between overriding and overloading?

- **Overriding** 🔄: Occurs when a subclass provides its own implementation of a method that is already defined in its superclass. The method signature must be the same. This is runtime polymorphism.
- **Overloading** 🔄: Occurs when you define multiple methods in the same class with the same name but different method signatures (number or type of parameters). This is compile-time polymorphism.

**Example:** An `EmailService` has an overloaded method `sendEmail(String recipient)` and `sendEmail(List<String> recipients)`. A subclass `MarketingEmailService` might then **override** the `sendEmail(String recipient)` method to add specific tracking features.

---

Can I override the overloaded method?

**Yes** ✅. A subclass can override a method from its parent, and it can also overload that overridden method by creating new methods with the same name but different parameters.

---

Can I override a private method?

**No** ❌. Private methods are not inherited and are not visible to subclasses, so they cannot be overridden.

---

Can I override the static method?

**No** ❌. Static methods belong to the class, not an instance. They are resolved at compile-time. If you define a static method with the same signature in a subclass, it's called **method hiding**, not overriding. The behavior depends on the reference type, not the object's actual type.

---

Can I overload the overridden method?

**Yes** ✅. You can overload any method in a class, whether or not it's also overriding a method from a parent.

---

Can I overload the final method?

**Yes** ✅. `final` methods cannot be overridden, but they can be overloaded with different parameters in the same class.

---

Can I declare a static method in interface?

**Yes** ✅. Since Java 8, interfaces can contain static methods. They are used for utility methods related to the interface itself and cannot be overridden by implementing classes.

**Example:** The `java.util.Comparator` interface has a static method `comparing()` that simplifies comparator creation.

---

Can I change the return type in an overridden method?

**Yes** ✅, since Java 5, you can change the return type to a **covariant return type**. This means the new return type must be a subtype of the original return type.

**Example:**

Java

```
class ReportGenerator {
    public Report createReport() { return new Report(); }
}
class FinancialReportGenerator extends ReportGenerator {
    @Override
    public FinancialReport createReport() { return new FinancialReport(); }
}
```

Here, `FinancialReport` is a subtype of `Report`.

---

Can I call a non-static method from a static method?

**Yes** ✅, but only by first creating an instance of the class. A static method belongs to the class and doesn't have an object instance to operate on.

**Example:** In a main method (which is static), to call a non-static `calculateTotal()` method from a `OrderService`, you would do:

```
public static void main(String[] args) {  
    OrderService service = new OrderService();  
    service.calculateTotal();  
}
```

What is the difference between composition and aggregation?

Both are "has-a" relationships, but they differ in ownership and lifecycle.

- **Composition** (strong relationship 🏠): The contained object cannot exist without the container. The lifecycle is tied. **Example:** A `Car` and its `Engine`. If the car is destroyed, the engine is too.
- **Aggregation** (weak relationship 🗑️): The contained object can exist independently. **Example:** A `Department` and its `Professors`. A professor can exist even if the department is dissolved.

What is the difference between abstraction and encapsulation?

- **Abstraction** 🏠: Hiding implementation details and showing only essential features. It focuses on *what* an object does. Achieved with abstract classes and interfaces.
- **Encapsulation** 📦: Bundling data and methods into a single unit (a class) and restricting direct access to the data. It's the mechanism that supports abstraction.

**Example:** A `PaymentGateway` interface is an **abstraction** that hides the complex logic of different payment providers. The `User` class with a private `password` field and a public setter method is an example of **encapsulation**.

What is polymorphism?

Polymorphism means "many forms". It's the ability of an object to take on many different forms. It allows a single interface to be used for a general class of actions. In Java, it's primarily achieved through method overriding.

**Example:** A payment processing system might have different payment types like `CreditCardPayment` and `PayPalPayment`. All of them implement a common `Payment` interface with a `process()` method. You can treat all of them as a generic `Payment` object and call `payment.process()`, and the correct implementation will be executed at runtime.

Why do we need 3 different classes for working with strings: `String`, `StringBuilder` and `StringBuffer`?

- `String` 🏠: An **immutable**, thread-safe class. Operations create new `String` objects. Ideal for scenarios where the string content should not change, like keys in a `HashMap`.
- `StringBuilder` 🗑️: A **mutable**, non-synchronized class. It's faster and more memory-efficient for frequent string modifications in a **single-threaded** environment.
- `StringBuffer` 🗑️: A **mutable**, synchronized, and therefore **thread-safe** class. Use it for string modifications in a multi-threaded environment where multiple threads might be appending to the same string simultaneously.

**Example:** Building a large HTML string from a database in a single-threaded application would use `StringBuilder` for performance. If a logger service appends log entries from multiple threads, it would use `StringBuffer` to prevent data corruption.

---

What is immutability?

Immutability is a design principle where an object's state **cannot be changed** after it is created. Any operation that appears to modify an immutable object actually returns a new object with the new state.

---

Is String immutable?

**Yes** . `String` is immutable. This is a core design decision in Java for thread safety and optimization.


---

How to implement an immutable object?

To create a truly immutable class:

1. Declare the class as `final` to prevent subclassing.
  2. Make all fields `private` and `final`.
  3. Do not provide any setter methods.
  4. For any mutable object fields (like a `List`), return a **defensive copy** from the getter to prevent external modification.
- 

Can we rethrow the same exception multiple times?



**Yes** . You can rethrow an exception from a `catch` block. This is often done to perform some logging or cleanup before propagating the exception up the call stack.

---

Describe the top three levels of exception hierarchy.

1. `java.lang.Object`: The root of all classes.
  2. `java.lang.Throwable`: The superclass of all errors and exceptions.
  3. `java.lang.Error` and `java.lang.Exception`: `Error` represents severe, unrecoverable problems (e.g., `OutOfMemoryError`), while `Exception` represents issues that can be handled by the application.
- 

What is the difference between checked and unchecked exceptions?

- **Checked Exceptions** : These are exceptions that the compiler forces you to handle (with `try-catch` or `throws`). They are for predictable but recoverable issues, like an `IOException`.
  - **Unchecked Exceptions** : These are not checked at compile time. They are for programming errors like `NullPointerException` or `IllegalArgumentException`. The developer is responsible for handling them, but the compiler doesn't force it.
- 

Is `Throwable` an interface or a class?

`Throwable` is a **class**. It is the superclass of all exceptions and errors in Java.

---

Can we use `try-finally` without `catch`?

**Yes** ✓. A `try` block can be followed by a `finally` block without a `catch`. This is used to guarantee that a cleanup action (like closing a file or a database connection) happens, whether an exception occurs or not.

---

In a method, `try {return 1} finally {return 2;}`. What will be returned?

The method will return `2`. The `finally` block's `return` statement overrides the `return` statement in the `try` block.

---

`try {throw new Exception();} catch(Exception ex){throw new Exception();}finally{...}`. Will "finally" be executed?

**Yes** ✓. The `finally` block is guaranteed to execute, even if an exception is thrown from the `try` and then another exception is thrown from the `catch`.

---

Why does `HashMap` use `LinkedList` inside it instead of `ArrayList`?

A `HashMap` uses a `LinkedList` (or a similar structure like a balanced tree in newer versions) to handle **hash collisions** ✨. When multiple keys hash to the same bucket index, the `LinkedList` chains them together. Using a `LinkedList` is more efficient for insertions and lookups in this chained structure than a dynamically-sized array like an `ArrayList`.

---

Can I call a private method outside class?

**No** ✗. `private` methods are only accessible within the class they are defined in.

---

What is a `transient` variable?

A `transient` variable is a field that should **not be serialized** when an object is written to a stream (e.g., to a file or sent over a network). When deserialized, its value will be the default for its type. **Example:** A user's password field should be marked `transient` to prevent it from being stored in a serialized form.

---

What is a "static" keyword?

The `static` keyword means a member (variable or method) **belongs to the class itself** rather than to any specific instance of the class. It is shared across all instances. **Example:** The `main` method is static, allowing it to be called without creating an instance of the class.

---

Is multiple inheritance supported in Java?

**No** ✗. Java does not support multiple inheritance of classes to avoid the "diamond problem". However, it does support **multiple inheritance of type** through interfaces.

---

Can I call more than one constructor from the constructor?

**No** ✗. You can only call one constructor from another constructor using `this()` or `super()`.

---

Can I override a constructor?

No **✗**. Constructors are not inherited by subclasses, so they cannot be overridden.

---

Can `hashCode()` return negative value?

Yes **✓**. The `hashCode()` method is allowed to return a negative integer. Hash-based collections like `HashMap` and `HashSet` handle negative values internally by converting them to a positive index.

---

GC algorithms and types of memory

- **GC Algorithms:**
    - **Serial GC:** Single-threaded, simple.
    - **Parallel GC:** Multi-threaded, high-throughput.
    - **G1 GC:** Region-based, low-pause time, good for large heaps.
    - **ZGC/Shenandoah:** Very low-pause time, for massive heaps.
  - **Types of Memory (Heap):**
    - **Young Generation:** For newly created objects.
    - **Old Generation:** For long-lived objects.
    - **Metaspace:** Stores class metadata.
- 

What will happen if a runtime exception is thrown from a static initialization block?

The class will **fail to initialize**, and a `java.lang.ExceptionInInitializerError` will be thrown. Any subsequent attempt to use or reference this class will result in the same error being thrown.

---

Complexity of getting an object from a hashmap (big O notation) in the worst case (when `hashCode` is a constant).

In the worst case, all objects will hash to the same bucket, turning the `HashMap` into a `LinkedList`. Retrieving an object would then require iterating through the entire list. The time complexity is **O(n)**, where n is the number of elements in the map.

---

`HashSet` vs `TreeSet`

Feature	<code>HashSet</code>	<code>TreeSet</code>
Order	Unordered	Sorted (natural or custom)
Performance	<b>O(1)</b> (on average) for add/remove/contains	<b>O(logn)</b> for add/remove/contains

Export to Sheets

Why would we use Optional class?

`Optional` was introduced in Java 8 to explicitly handle the possibility of a value being absent, helping to prevent `NullPointerException` 🐛. It provides a clear, functional way to handle nulls, making the code more readable and robust.

**Example:**

Java

```
Optional<User> user = findUserId(123);  
user.ifPresent(u -> System.out.println(u.getEmail())); // No NullPointerException risk
```

## ☕ Java 8

Why were default methods in the interface added in Java 8?

**Default methods** were added to allow for the **evolution of interfaces without breaking existing implementations**. Before Java 8, adding a new method to an interface would require every single class that implemented it to be updated, which was not feasible for large codebases. A default method provides a default implementation that implementing classes can inherit without modification.

**Example:** The `java.util.List` interface added a `stream()` default method, which all existing implementations like `ArrayList` inherited automatically.

What is the difference between Stream and List?

- **List**: A data structure 📦. It holds elements in memory and is used for storing and accessing data.
- **Stream**: A sequence of elements for processing ⚙️. It is not a data structure and does not store elements. Streams are lazy and designed for functional-style operations like `filter`, `map`, and `reduce`.

What are intermediate methods in streams?

**Intermediate methods** are stream operations that return a new stream. They are **lazy** and do not execute until a terminal operation is called. They are used to transform or filter the stream.

**Examples:** `filter()`, `map()`, `sorted()`.

What can't be done with parallel streams?

Parallel streams are generally not suitable for operations that depend on order or have shared state. Avoid using them for:

- Stateful lambdas.



- Operations that rely on the order of elements (e.g., using `forEach` instead of `forEachOrdered`).
- Operations with a non-associative reduction function.

Describe three main parts of lambda expressions

A lambda expression has three main parts:

1. **Parameters:** The input parameters, enclosed in parentheses. (e.g., `(a, b)`)
2. **Arrow Token:** The `->` token, which separates parameters from the body.
3. **Body:** The expression or code block that defines the behavior. (e.g., `return a + b;` or `System.out.println("Hello");`)

Main features of new DateTime API

- **Immutability:** All `DateTime` objects are immutable and thread-safe.
- **Clarity:** Separates date, time, and timezone concepts into different classes (`LocalDate`, `LocalTime`, `ZonedDateTime`).
- **Fluent API:** Method chaining for easy manipulation.
- **Thread-safe:** Designed for concurrent environments.

## Multithreading


What is the difference between Thread and Process?

Feature	Process	Thread
Memory	Has its own separate memory space.	Shares memory space with other threads in the same process.
Execution	An independent program.	A unit of execution within a process.
Creation	Heavyweight.	Lightweight.

Export to Sheets

**Analogy:** A **process** is a separate browser application. A **thread** is a tab within that browser. All tabs share the browser's memory but perform different tasks.

What is the difference between `Thread.start()` and `Thread.run()`?

- `Thread.start()`  : Creates a **new thread** and then executes the `run()` method in that new thread. This

is how you achieve concurrency.

- `Thread.run()` 🏃 : Simply executes the `run()` method in the current thread. It's just a regular method call. No new thread is created.

---

What is the purpose of a `volatile` keyword?

The `volatile` keyword ensures that a variable's value is always read from and written to **main memory**, not from the thread's local cache. This guarantees **visibility** of the variable's value across multiple threads. It does not provide atomicity.

---

What is semaphore?

A **semaphore** is a synchronization primitive that controls access to a shared resource by a limited number of threads. It maintains a count of available permits. A thread must acquire a permit to access the resource and release it when done. **Example:** A connection pool might use a semaphore to limit the number of active database connections.

---

What is the difference between `sleep` and `wait`?

Feature	<code>sleep()</code> (from <code>Thread</code> )	<code>wait()</code> (from <code>Object</code> )
Lock Release	<b>Does not</b> release the lock.	<b>Releases</b> the lock on the object.
Called from	Anywhere.	Must be called from a <code>synchronized</code> block.
Waking up	Wakes up after a specified time.	Wakes up when <code>notify()</code> or <code>notifyAll()</code> is called.

---

Export to Sheets

---

How can I get an exception from some other thread?

You cannot directly "catch" an exception from another thread. You can handle exceptions from worker threads by using:

- A `Callable` with an `ExecutorService`: The exception is wrapped inside the `Future` object and re-thrown when you call `future.get()`.
- An `UncaughtExceptionHandler`: A special handler that can be set on a thread to deal with any uncaught exceptions.

---

What happens when an exception is thrown from a thread and it doesn't have an exception handler?

The thread will **terminate**, and the JVM will print a stack trace to the console. The rest of the application will

continue to run unless it was a fatal error for the entire JVM.

---

What states of Thread do you know?

1. **NEW**: The thread has been created.
  2. **RUNNABLE**: The thread is executing or ready to execute.
  3. **BLOCKED**: The thread is waiting for a monitor lock.
  4. **WAITING**: The thread is waiting indefinitely for another thread to perform a certain action.
  5. **TIMED\_WAITING**: The thread is waiting for a specified amount of time.
  6. **TERMINATED**: The thread has finished execution.
- 

A thread has been started and then finished their work. Can I start this thread again?

No **✗**. A thread can only be started once. Once it enters the **TERMINATED** state, you must create a new **Thread** object to perform work again.

---

How to implement thread safe singleton?

One of the most common and robust ways is using **Double-Checked Locking** with a **volatile** field.

Java

```
public class MySingleton {
    private static volatile MySingleton instance;
    private MySingleton() {}
    public static MySingleton getInstance() {
        if (instance == null) {
            synchronized (MySingleton.class) {
                if (instance == null) {
                    instance = new MySingleton();
                }
            }
        }
        return instance;
    }
}
```

How many stacks do we have in JVM?

Each **thread** in the JVM has its own private **Java Virtual Machine Stack**. This stack is used to store local variables and method call information.

---


**Future** vs **Callable**

- **Callable**: An interface representing a task that can **return a value** and **throw a checked exception**.
- **Future**: An object that represents the **result of an asynchronous computation**. You can use a **Future** to check if a **Callable** task is complete, wait for it, and retrieve its result.

What are possible scopes of the bean?

- `singleton`: (Default) One instance per container.
  - `prototype`: A new instance every time it is requested.
  - `request`: A new instance for each HTTP request.
  - `session`: A new instance for each HTTP session.
  - `application`: A single instance for the entire `ServletContext`.
  - `websocket`: A single instance per WebSocket session.
- 

Is it possible to have two singleton beans of the same type in the same spring context?

Yes . You just need to give them different bean names using `@Bean(name = "myBean1")` and `@Bean(name = "myBean2")` or by specifying a qualifier.

---

What will happen if we inject a prototype scope bean into singleton scope bean?

The prototype bean will be instantiated **only once** when the singleton bean is created and injected into it. The singleton will hold a reference to this single instance, defeating the purpose of the prototype scope.

---

I create a singleton bean and autowire a prototype bean into it. After that I call the singleton bean 10 times. How many prototype beans will be created? How to do that prototype bean is created for each call of singleton.


- Only **one** prototype bean is created and injected.
  - To create a new prototype bean for each call, you must use **Method Injection**. The singleton bean should inject an `ObjectFactory` or the `ApplicationContext`, and then programmatically request a new instance each time.
- 

How this situation can occur: I annotate one bean as a singleton scope bean, after that start application and in VisualVM I see that there are 2 instances of this bean?

This can happen if:

1. There are **two separate Spring contexts** within the application (e.g., a root context and a servlet context).
  2. A class is manually instantiated using `new`, bypassing the Spring container entirely.
- 

All possible ways to inject dependencies in Spring.

1. **Constructor Injection** (Recommended ): Dependencies are provided through the class constructor.  
Java

```
public MyService(MyDependency dep) { ... }
```

1. **Setter Injection**: Dependencies are injected via a public setter method.

Java

```
@Autowired
public void setMyDependency(MyDependency dep) { ... }
```

1. **Field Injection** (Not recommended ⚠️): Dependencies are injected directly into a field.

Java

```
@Autowired
private MyDependency dep;
```

---

How to inject a dependency from a non-spring context?

You can manually create the object and register it as a bean in a `@Configuration` class:

Java

```
@Configuration
public class MyConfig {
    @Bean
    public ExternalService externalService() {
        return new ExternalService();
    }
}
```

---

How to use a bean from a spring context outside of the spring application?

You can retrieve a reference to the `ApplicationContext` and use its `getBean()` method. This is generally discouraged as it creates a tight coupling to the Spring framework.

---

Are beans thread-safe?

By default, **Spring beans are singletons and are not inherently thread-safe ❌**. You must ensure thread safety yourself by making the beans stateless or by using proper synchronization.

---

What is the difference between `BeanFactory` and `ApplicationContext`?

- `BeanFactory`: The basic Spring container. It provides fundamental dependency injection and is lazy-loading.
- `ApplicationContext`: A more advanced container with additional features like AOP, event publishing, and internationalization. It is typically eager-loading (creates singleton beans on startup). `ApplicationContext` is the preferred choice for most applications.

---

What does the Spring bean lifecycle look like?

1. **Instantiation** 🤖 : The bean is created.
  2. **Population of properties** 🖋️ : Dependencies are injected.
  3. **Initialization** ✨ : `BeanPostProcessors` are called, followed by custom init methods like `@PostConstruct`.
  4. **Ready for use** ✅ : The bean is in the container, ready to be used.
  5. **Destruction** 💥 : The bean is destroyed when the container is closed, and custom destroy methods like `@PreDestroy` are called.
- 

What is a Spring Boot?

**Spring Boot** is a convention-over-configuration framework that simplifies the creation of stand-alone, production-grade Spring applications. It provides auto-configuration, an embedded web server (like Tomcat), and opinionated starter dependencies, reducing boilerplate code.

---

Name some of the Design Patterns used in the Spring Framework?

- **Singleton**: The default scope for beans.
  - **Factory**: `BeanFactory` and `ApplicationContext` are examples of a Factory.
  - **Proxy**: Used for AOP.
  - **Template Method**: Used in classes like `JdbcTemplate`.
  - **Observer**: For Spring's event handling.
- 

What is the role of the `@Autowired` annotation?

The `@Autowired` annotation tells Spring to **automatically inject a bean** into a field, setter method, or constructor.

---

How to handle exceptions in Spring MVC environment?

You can handle exceptions using:

1. `@ExceptionHandler` 🖋️ : A method-level annotation to handle specific exceptions within a single controller.
  2. `@ControllerAdvice` 🖋️ : A class-level annotation for global exception handling across multiple controllers.
  3. `ResponseStatusException` : To handle simple cases.
- 

Name some standard Events in spring framework

- `ContextRefreshedEvent` : Published when the `ApplicationContext` is initialized or refreshed.
  - `ContextStartedEvent` : Published when the `ApplicationContext` is started.
  - `ContextStoppedEvent` : Published when the `ApplicationContext` is stopped.
  - `ContextClosedEvent` : Published when the `ApplicationContext` is closed.
- 

Integration testing / integration context in spring

Integration testing in Spring involves loading a small, specific subset of the Spring application context to test the interaction between different layers (e.g., a controller and a service). This is typically done using

`@SpringBootTest` or `@ContextConfiguration`.

---

Transaction Management ( `@Transactional` )

The `@Transactional` annotation provides a declarative way to manage transactions. When applied to a method, a proxy is created that manages the transaction for that method, ensuring that all database operations are treated as a single, atomic unit.

---

How to work with two databases in one application?

To work with two databases, you need to configure two separate `DataSource` beans, two `EntityManagerFactory` beans, and two `TransactionManager` beans. You then use `@Qualifier` annotations to specify which bean to use for specific repositories or services.

---

How is it possible that jars inside an “uber jar” appear in the class path?

Spring Boot creates an executable "uber jar" with a special launcher class ( `JarLauncher` ). When you run the jar, this launcher is the first class that executes. It programmatically adds the nested jars inside the `BOOT-INF/lib` directory to the application's classpath using a custom class loader.

---

## Hibernate

---

Is `Session` object thread-safe?

**No** ❌. A `Session` object is **not thread-safe**. A new `Session` should be opened for each thread or for each logical transaction. Sharing a `Session` between threads can lead to unpredictable behavior and concurrency issues.

---

It is necessary to run three queries in parallel with hibernate. Will it require three session objects or one session object will be enough?

It requires **three separate** `Session` objects, one for each thread.

---

How to map one entity to two tables?

You can map an entity to multiple tables using the `@SecondaryTable` annotation. This allows you to specify a second table for an entity, and you can map fields to this table using the `@Column` annotation with the `table` attribute.


---

What is a first-level cache?

The **first-level cache** is a mandatory cache associated with a Hibernate `Session` object. Hibernate stores all objects loaded or saved within that specific session in this cache. If the same object is requested multiple times within the same session, it's returned from the cache without a database round trip.

---

Explain all states of entity.

1. `Transient` : An object is in this state when it's just been created with `new` and is not associated with

any `Session`.

2. **Persistent** 🟢: The object is associated with a `Session`. Changes to the object are automatically detected by Hibernate (dirty checking).
  3. **Detached** ⏸️: The object was once persistent, but its `Session` has been closed. Changes will not be synchronized to the database.
  4. **Removed** 🗑️: The object is marked for deletion. It will be removed from the database when the transaction is committed.
- 

Explain n+1 problem

The **N+1 problem** is a performance anti-pattern where Hibernate executes one query to retrieve a collection of parent entities (the "1" query) and then `N` separate queries to retrieve the lazy-loaded child collections for each parent.

**Solution:** Use `join fetch` in HQL/JPQL or `@Fetch(FetchMode.SUBSELECT)` to load all related data in a single query.

---

How to implement distributed cache in Hibernate?

You can implement a distributed cache by configuring a **second-level cache** provider like **Redis** or **Hazelcast** and setting up a distributed caching topology.

---

Different types of fetching

- **Lazy Fetching** 🐢: (Default for collections) Associated data is loaded only when it is accessed.
  - **Eager Fetching** ⚡: Associated data is loaded immediately with the main entity.
- 

What is the difference between the Hibernate `Session.get()` and `load()` method?

- `session.get()`: Fetches the object from the database immediately. Returns `null` if the object is not found.
  - `session.load()`: Returns a **proxy** (a placeholder) immediately. It only hits the database when a non-ID property is accessed. Throws an `ObjectNotFoundException` if the object doesn't exist.
- 

What is automatic dirty checking in Hibernate?

**Dirty checking** is a feature that automatically detects any changes made to a persistent object and synchronizes those changes to the database when the `Session` is flushed or the transaction is committed, without requiring an explicit `update()` call.

---

What is Query Cache in Hibernate?

The **Query Cache** is a caching mechanism that stores the results of queries. When the same query is executed again with the same parameters, Hibernate returns the result from the cache.

---

What are the inheritance mapping strategies?



1. **Single Table**: All classes in the hierarchy are stored in a single table with a "discriminator column" to identify the subclass.
2. **Joined Table**: Each class has its own table, and data is joined across tables to retrieve a full object.
3. **Table per Class**: Each concrete class has its own table, which duplicates parent fields.

---

What will happen if a lazy loaded entity is accessed in a detached state?

A **LazyInitializationException** will be thrown. When the entity is detached, its **Session** is closed, and Hibernate can no longer fetch the data for the lazy-loaded field.

---

When is an id set for an entity?

The ID is set for an entity when it is **saved or persisted** by Hibernate. The exact timing depends on the ID generation strategy. For **GenerationType.IDENTITY**, the ID is set after the **INSERT** query. For **GenerationType.SEQUENCE**, the ID is set before the **INSERT**.

---

In what cases **equals()** and **hashCode()** should be overridden when using Hibernate?

You should override **equals()** and **hashCode()** for entities to ensure they function correctly in collections like **HashSet**. The implementation should be based on a **business key** (a unique, immutable field) rather than the primary key, as the primary key may not be set for a transient object.

---

Lazy loading collection was not loaded but the session was closed. How to load this collection?

You cannot load a lazy collection once the session is closed. To load it, you must either:

- Change the fetching strategy to **EAGER**.
- Use **Hibernate.initialize(collection)** on the collection while the session is still open.
- Use a **join fetch** in your query to load the collection in the same query as the main entity.

## Databases

---

Explain all types of joins

Type	Description
<b>INNER JOIN</b>	Returns rows with matching values in both tables.
<b>LEFT JOIN</b>	Returns all rows from the left table and the matched rows from the right.
<b>RIGHT JOIN</b>	Returns all rows from the right table and the matched rows from the left.

<b>FULL JOIN</b>	Returns rows when there is a match in one of the tables.
------------------	--

Export to Sheets

What is an index?

An **index** is a data structure (like a B-tree) that improves the speed of data retrieval operations on a database table. It provides a quick lookup for a specific column value without scanning the entire table.

Why don't we use an index for every column?

Indexes add overhead. They require **extra storage space**, and they **slow down** **INSERT**, **UPDATE**, and **DELETE** operations because the index must also be updated. You should only use indexes on columns that are frequently used in **WHERE**, **ORDER BY**, or **JOIN** clauses.

Explain Isolation levels

Isolation levels define how and when changes made by one transaction become visible to other concurrent transactions.

- READ UNCOMMITTED**: Allows "dirty reads" (reading uncommitted data).
- READ COMMITTED**: Prevents dirty reads.
- REPEATABLE READ**: Prevents dirty reads and non-repeatable reads.
- SERIALIZABLE**: The highest level, which prevents all concurrency issues but with a significant performance penalty.

What's the difference between SQL & NoSQL databases?

Feature	SQL (Relational)	NoSQL (Non-relational)
<b>Schema</b>	Fixed, predefined schema.	Dynamic schema.
<b>Data Model</b>	Tables with rows and columns.	Documents, key-value pairs, graphs, etc.
<b>Scaling</b>	Vertical scaling (up).	Horizontal scaling (out).
<b>Transactions</b>	ACID-compliant.	BASE-compliant.

Export to Sheets

How to find the number of customers in each location? (`TABLE customer(name, location_id)`)

SQL

```
SELECT location_id, COUNT(*)FROM customerGROUP BY location_id;
```

How to implement MANY-TO-MANY relationships?

You implement a many-to-many relationship using a `JOIN` or `LINKING` table. This table contains foreign keys that link to the primary keys of the two tables in the relationship.

**Example:** A `students` and `courses` table would have a `student_courses` join table with `student_id` and `course_id`.

What we will get if we execute `Select * From A, B`? (`TABLE A (a), TABLE B(b)`)

This is a `CROSS JOIN`. It produces a **Cartesian product** of the two tables, returning every row from table `A` combined with every row from table `B`.

There is a table with duplicates. How to delete duplicates?

You can delete duplicates by using a `DELETE` statement with a subquery that identifies duplicate rows.

**Example:**

SQL

```
DELETE FROM your_tableWHERE id NOT IN (  
    SELECT MIN(id) FROM your_table GROUP BY col1, col2  
);
```

What is a prepared statement?

A **prepared statement** is a pre-compiled SQL statement used to execute the same or similar queries repeatedly. It has placeholders (`?`) for parameters, which are sent to the database separately from the SQL.

What are the benefits of using prepared statements?

- **Performance:** The database compiles the query plan only once.
- **Security:** Prevents **SQL injection attacks** by separating the SQL logic from the user-provided data.

When would we use “having” key-word in an SQL query?

The `HAVING` clause is used to **filter rows after** a `GROUP BY` clause has been applied. It operates on the grouped result set, whereas a `WHERE` clause filters rows before they are grouped.

**Example:** `SELECT location_id, COUNT(*) FROM customer GROUP BY location_id HAVING COUNT(*) > 10;`

What is JAX/RS?

**JAX-RS (Java API for RESTful Web Services)** is a Java specification that provides a standardized set of APIs and annotations for developing RESTful web services.

---

What is Resource in REST?

A **Resource** is an abstraction of a piece of information, identified by a unique URI. It is the core concept of REST. You interact with resources using standard HTTP methods.

**Example:** The URI `/users/123` identifies a specific user resource.

---

What is the difference between `GET` and `POST` requests?

- **GET**: Used to **retrieve** data. It is **safe** (no side effects) and **idempotent**. Data is in the query string.
  - **POST**: Used to **submit** data to create or update a resource. It is **not idempotent**. Data is in the request body.
- 

What are the parts of HTTP request?

1. **Request Line:** `GET /users HTTP/1.1`
  2. **Headers:** Key-value pairs providing metadata (e.g., `Content-Type`).
  3. **Body:** The data being sent (optional).
- 

How can you send arguments via REST?

- **Path variables:** `/users/{id}`
  - **Query string:** `/users?status=active`
  - **Request body:** JSON or XML data for `POST` and `PUT`.
  - **Headers:** For metadata like authentication tokens.
- 

How to say to server, that it should work with XML, not with JSON

You can use the **Accept** header in the HTTP request. Setting `Accept: application/xml` tells the server that the client wants a response in XML format.

---

How can we make REST endpoints secure?

- **HTTPS:** To encrypt communication.
  - **Authentication:** To verify the client's identity (e.g., OAuth2, JWT).
  - **Authorization:** To grant access based on user permissions.
  - **Input Validation:** To prevent injection attacks.
- 

What is content type negotiation?

**Content type negotiation** is the process where a client and server agree on the data format for communication. The client sends an `Accept` header to specify its preferred formats, and the server returns a `Content-Type` header indicating the chosen format.

---

What is an idempotent operation?

An **idempotent operation** is one that produces the same result regardless of how many times it is executed.

`GET`, `PUT`, and `DELETE` are considered idempotent. `POST` is not.

---

How to implement stateful service using REST?

REST is fundamentally stateless. To maintain state, you manage it on the client side by having the client send a state identifier (like a session ID or token) with each request. The server uses this identifier to retrieve the corresponding state from a central store (e.g., a database or cache).

## Design Patterns

---

SOLID with examples from Java

- **S - Single Responsibility Principle:** A class should have only one reason to change. **Example:** Separate `UserService` from `EmailService`.
  - **O - Open/Closed Principle:** Open for extension, closed for modification. **Example:** Add new payment types by extending a `PaymentProcessor` class without modifying its core.
  - **L - Liskov Substitution Principle:** A subclass should be substitutable for its superclass. **Example:** If `Duck` has a `fly()` method, a `RubberDuck` subclass shouldn't implement a no-op `fly()` because it can't fly.
  - **I - Interface Segregation Principle:** Clients should not be forced to depend on interfaces they do not use. **Example:** Use smaller, specific interfaces like `SmsService` and `EmailService` instead of a single `NotificationService`.
  - **D - Dependency Inversion Principle:** High-level modules should not depend on low-level modules. Both should depend on abstractions. **Example:** A `OrderService` depends on a `PaymentProcessor` interface, not a concrete `CreditCardPaymentProcessor` class.
- 

Describe Singleton pattern

The **Singleton pattern** ensures that a class has only **one instance** and provides a global point of access to it. It's often used for a database connection pool or a configuration manager.

---

What is Chain of Responsibility pattern

The **Chain of Responsibility** pattern allows an object to send a request to a chain of handlers. Each handler decides whether to process the request or pass it to the next handler in the chain. It decouples the sender from the receiver.

**Example:** A payment request might go through a chain of handlers: a `ValidationHandler`, then a `FraudHandler`, then a `ProcessingHandler`.

---

Difference between Factory and Builder patterns

- **Factory** 🏭 : Creates objects without exposing the instantiation logic. Good for creating a single object in a simple creation process.
  - **Builder** 🏗️ : Constructs a complex object step-by-step. Good for objects with many optional parameters, providing a fluent API.
- 

“Template method” pattern.

The **Template Method** pattern defines the skeleton of an algorithm in a base class but allows subclasses to override certain steps of the algorithm without changing the overall structure.

**Example:** The `java.util.AbstractList` class defines a `get()` method as a template, leaving the actual data retrieval to be implemented by concrete subclasses like `ArrayList` and `LinkedList`.

---

Your opinion on if composition is better than inheritance.

**Composition is generally preferred over inheritance.** Inheritance creates a rigid "is-a" relationship and tight coupling, making it difficult to change or extend class hierarchies. Composition creates a more flexible "has-a" relationship, promoting loose coupling and better code reuse.

---

There are two classes `class A {void a();}` and `class B {void b();}` How to reuse methods `a()` and `b()` in a new class?

Use **composition**. Create a new class, say `ClassC`, that has instances of `ClassA` and `ClassB` as its fields. You can then call `a()` and `b()` on those instances. This avoids the problems of multiple inheritance.

---

Functional vs Imperative programming.

- **Imperative** 📝 : Focuses on **how to achieve a result** by providing a sequence of commands that change the program's state. **Example:** A `for` loop that iterates and adds elements to a list.
  - **Functional** ⚙️ : Focuses on **what the result should be** by using expressions and functions. It avoids mutable state and side effects. **Example:** Using a Java Stream with `filter()` and `map()`.
-