New chat
Gems
Structure
Explore Gems
Recent
Core Java Concepts Explained
Open Closed Principle in Agile
Angular Admin Link Navigation Tr...
Fix Angular Admin Review Routing
Temporarily Disable JPA on Laptop
Angular Package Management an...
A Language Model's Identity
Understanding the Design Thinke...
Platform Diagram for Skills Analy...

1. What is the difference between JVM, JRE and JDK.
2. Explain the architecture of JVM.
3. How garbage collector works in java.

Sure! This is a comprehensive set of questions covering core Java concepts. Here are the answers:

1. JVM, JRE, and JDK

Acronym	Full Form	Description
JVM	Java Virtual Machine	An abstract machine that provides the runtime environment to execute Java bytecode. It's the core component.
JRE	Java Runtime Environment	The set of software tools that includes the JVM and the Java Class Libraries needed to run compiled Java applications. It's what you need to **run** a Java program.
JDK	Java Development Kit	A superset of JRE that also includes development tools like the **Java compiler** (`javac`), debugger, and documentation. It's what you need to **develop, compile, and** run Java programs.

Export to Sheets

2. Architecture of JVM

The JVM architecture is composed of three main subsystems:

1. **Class Loader Subsystem:**
   - **Loading:** Finds and loads the binary representation of a class file (`.class` file) based on its fully qualified name.
   - **Linking:**
     - **Verification:** Ensures the correctness and security of the bytecode.
     - **Preparation:** Allocates memory for static variables and initializes them to default values.
     - **Resolution:** Replaces symbolic references in the runtime constant pool with direct references.
   - **Initialization:** Executes the class's static initializers and initializes static variables to their proper values.
2. Runtime Data Area (Memory Areas)
   - **Method Area:** Stores class structures like runtime constant pool, field and method data, and the code for methods. Shared among all threads.
   - **Heap:** Stores all **objects** and their corresponding instance variables and arrays. Shared among all threads. This is where the `new` keyword allocates memory.
   - **Stack (JVM Stacks):** Used for execution of a thread. Stores **local variables**, partial results, and data for method calls (each method call pushes a **Stack Frame** onto the stack). Each thread has its own private stack.
   - **PC Registers:** Stores the address of the currently executing JVM instruction. Each thread has its own private PC register.
   - **Native Method Stacks:** Used to hold native method information (methods written in languages other than Java, e.g., C/C++). Each thread has its own private native method stack.
3. **Execution Engine**
   - **Interpreter:** Reads and executes bytecode instruction by instruction.
   - *JIT (Just-in-Time) Compiler:* Compiles frequently executed bytecode into native machine code for faster execution.
   - **Garbage Collector (GC):** Automatically manages memory (heap) by destroying unreachable objects.

3. How Garbage Collector Works in Java

The **Garbage Collector (GC)** in Java is an automatic memory management process that frees up memory occupied by **unreachable objects** (objects that no longer have any active references).

1. **Identify Unreachable Objects:** GC starts by identifying root objects, called **GC Roots** (e.g., local variables, static variables, active threads). Any object that cannot be reached by traversing the object graph starting from these roots is considered **eligible for garbage collection**.
2. **Collection Process:** Once an object is identified as garbage, the GC reclaims the memory it occupies.
3. **Heap Segments:** Modern GCs divide the **Heap** into generations (e.g., **Young Generation** for newly created objects, **Old/Tenured Generation** for long-lived objects) to improve performance, based on the **Weak Generational Hypothesis** (most objects die young).

4. Mark and Sweep Algorithm

Mark and Sweep is the most basic and common garbage collection algorithm used for finding and reclaiming memory:

1. **Mark Phase:** The GC traverses the object graph starting from the GC Roots and marks all objects that are **reachable** (in use). This is typically done by setting a flag bit on the object's header.
2. **Sweep Phase:** The GC scans the entire heap, looking for objects that were not marked in the Mark phase. These unmarked objects are **unreachable** (garbage). The memory occupied by these objects is reclaimed and added to a free list.

A major drawback of Mark and Sweep is **fragmentation** (small, scattered free memory holes), which can be addressed by subsequent compaction.

5. Memory Allocation using `new`

When the `new` keyword is used to allocate memory to an object, the memory is allocated on the **Heap** (specifically, in the **Young Generation** area of the Heap for modern GCs).

- **Instance variables** for that object are stored on the Heap.
- The **reference variable** pointing to the object is typically stored on the **Stack**.

6. Different Kinds of Garbage Collectors

Java has evolved with several major types of Garbage Collectors, which often fall into two main categories:

1. **Serial GC:** Uses a single thread to perform all GC work. It causes "Stop-The-World" pauses for collection. Suitable for single-processor machines or applications with very small heaps.
2. **Parallel GC (Throughput Collector):** Uses multiple threads for collection in the Young and Old generations. Designed to maximize application throughput. It also causes "Stop-The-World" pauses.
3. **Concurrent Mark-Sweep (CMS) GC:** Attempts to do most of its work **concurrently** with the application threads to minimize "Stop-The-World" pauses. It is deprecated since Java 9 and removed in Java 14.
   - `G1 Garbage Collector` (`-XX:+UseG1GC`) is the default collector since Java 9. It is a generational, garbage-first, and concurrent collector that divides the heap into regions and aims to meet a configurable pause time goal by prioritizing (Garbage-First) collection of regions with the most reclaimable memory.
4. **ZGC** and **Shenandoah GC:** Low-latency GCs designed for very large heaps (terabytes) and extremely short (sub-millisecond) pause times, even when performing most work concurrently.

7. The Use of `finalize()`

The `finalize()` method was deprecated since Java 9, and removed in Java 18 was a method in the `Object` class.

- **Purpose:** It was called by the Garbage Collector on an object just **before** the object was destroyed/reclaimed. It was intended for performing **cleanup activities** like closing files or external resources that were not managed by the JVM's garbage collection (e.g., native resources).
- **Drawbacks:** The order of finalization was not guaranteed, it could delay garbage collection, and it introduced performance overhead.
- **Modern Alternative** instead of `finalize()`, modern Java strongly recommends using `try-with-resources` for automatic resource management and the `Cleaner` API for complex object cleanup.

8. The Topmost Class of Java

The topmost class in the entire Java class hierarchy is `java.lang.Object`. All classes in Java implicitly inherit from this class.

9. Methods of Object Class

Since `java.lang.Object` is the root of the class hierarchy, its methods are inherited by every class. The primary methods are:

Method	Purpose
`Object clone()`	Creates and returns a copy of the object.
`equals(Object obj)`	Indicates whether some other object is "equal to" this one.
`void finalize()`	Called by the GC before object destruction (deprecated).
`Class<?> getClass()`	Returns the runtime class of this `Object`.
`int hashCode()`	Returns a hash code value for the object.
`void notify()`	Wakes up a single thread that is waiting on this object's monitor.
`void notifyAll()`	Wakes up all threads that are waiting on this object's monitor.
`String toString()`	Returns a string representation of the object.
`void wait()`	Causes the current thread to wait until another thread invokes `notify()` or `notifyAll()` on this object. (Has overloads for timeout).

Export to Sheets

10. `hashCode()` and `equals()`

`hashCode()`

The `hashCode()` method returns an **integer hash code value** for an object. It is used primarily in hash-based collections like `HashMap`, `HashSet`, and `Hashtable` to quickly determine a **bucket/location** where the object should be stored.

`equals(Object obj)`

The `equals()` method is used to determine if two objects are **logically equal** (represent the same data or state). By default, the `Object` class implementation performs a **reference comparison** (`this == obj`).

Why Override Them?

You should **always** override both `hashCode()` and `equals()` together if you intend to compare two objects based on their **contents/value** rather than their memory address (reference).

The core contract is:

- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.
- It is not required that if two objects have the same hash code, they must be equal.

Benefit of Overriding

Overriding them correctly ensures that your custom objects behave as expected in Java Collections:

- **Correctness in Sets:** If you use your objects in a `HashSet`, two logically equal objects will be treated as one single element (as expected).
- **Correctness in Maps:** If you use your objects as keys in a `HashMap`, you can reliably retrieve the value using a key that is logically equal to the one used for insertion.
- **Efficiency:** A well-distributed `hashCode()` implementation dramatically improves the performance of hash-based collections by minimizing collisions and speeding up object lookups.

11. Use of `clone()` of Object Class

The `clone()` method of the `Object` class is used to create a field-for-field copy of an existing object, it performs a **shallow copy**, meaning:

- **Primitive Fields:** The values of primitive fields are copied directly.
- **Object Reference Fields:** Only the **reference** (memory address) to the actual object is copied, not the referenced object itself. The original and cloned objects will share the same underlying referenced object.

To use `clone()`, the class must implement the `java.lang.Cloneable` marker interface (see Q14).

12. Use of `wait()`, `notify()`, `notifyAll()`

These methods are essential tools for **inter-thread communication** (synchronization) in Java. They must **only** be called from a synchronized context (a synchronized block or method) and operate on an object's **monitor** (intrinsic lock).

Method	Purpose
`wait()`	Causes the current thread to **release the lock** on the object and enter a waiting state until another thread calls `notify()` or `notifyAll()` on that object.
`notify()`	Wakes up **a single** thread that is waiting on the object's monitor. Which thread is chosen is arbitrary.
`notifyAll()`	Wakes up **all** threads that are waiting on the object's monitor. The threads then compete for the lock once the notifying thread releases it.

Export to Sheets

Key Concept: These methods facilitate the **Producer-Consumer** pattern and other coordinated concurrency models.

13. Use of `hashCode()`

The `hashCode()` method is used to get a **string representation** of an object.

- **Default Implementation:** The default implementation in `Object` returns a string consisting of the class name, followed by the '@' sign, followed by the hexadecimal representation of the object's hash code (e.g., `ClassName@1a2b3c4d`).
- Typical Use: It is almost always **overridden** in custom classes to provide a meaningful, human-readable description of the object's state (e.g., `Person[name=John, age=30]`).
- **Implicit Calls:** `toString()` is often called implicitly by Java when an object is concatenated with a string (e.g., `System.out.println("Object: " + myObject);`).

14. Java Reflection

Java Reflection is the ability of a program to **examine, inspect, and modify** its own structure and behavior at **runtime**. It allows you to:

- Inspect a class's members (fields, methods, constructors) and modifiers.
- Instantiate objects and invoke methods dynamically (even private ones).

Use of `getClass()` in Reflection

The `getClass()` method (from `Object`) `getClass()` is **the entry point** for reflection. It returns a `Class` object that represents the runtime class of the object. Once you have the `Class` object, you can use the methods of the `Class` class (e.g., `getDeclaredMethods()`, `getFields()`, `getConstructors()`) to perform reflection operations.

Sec 47.11 Java Class Information

47.11 Java Class Information

47.11 Java Class Information

47.11 Java Class Information

47.11

You cannot directly sort records using a `HashMap`.

The Problem

A `HashMap` uses a **hash table** structure and does **not** guarantee any order of elements (neither insertion order nor sorted order). Iterating over a `HashMap` produces an order that may seem random and can change over time.

Solutions (Using Alternative Map Types)

To store key-value pairs and maintain a sorted or predictable order, you must use different Map implementations:

1. **Sort By Key:** Use a `TreeMap`.
   - A `TreeMap` stores its entries in a natural sorting order of the keys, or according to a custom `Comparator` provided at creation time.

   Java

   ```
   // Sorts by the key's natural order (e.g. String/Integer)
   Map<String, Integer> sortedMap = new TreeMap<>();
   ```

2. **Sort By Insertion Order:** Use a `LinkedHashMap`.
   - A `LinkedHashMap` maintains a doubly-linked list of its entries, ensuring that iteration order is the order in which entries were inserted (or optionally, last accessed).

   Java

   ```
   Map<String, Integer> insertionOrderMap = new LinkedHashMap<>();
   ```

3. **Sort By Value:** This is more complex and requires manual manipulation:
   - Convert the `HashMap`'s entries into a `List` of `Map.Entry` objects.
   - Use `Collections.sort()` or `List.sort()` on this list, providing a custom `Comparator` that compares the entries based on their values.
   - Create a new `LinkedHashMap` and populate it from the sorted list of entries to maintain order.

   Java

   ```
   // Example for sorting by value (Descending)
   List<Map.Entry<String, Integer>> entryList = new ArrayList<>(myMap.entrySet());
   List.sort(Map.Entry.comparingByValue(Collections.reverseOrder));

   Map<String, Integer> sortedMap = new LinkedHashMap<>();
   for (Map.Entry<String, Integer> entry : entryList) {
       sortedMap.put(entry.getKey(), entry.getValue());
   }
   ```

1. What is Data Hiding. How do you achieve it Java.
2. What is data encapsulation.
3. What is data abstraction...

Here are the answers to your questions on core Java concepts.

1. Data Hiding

Data Hiding is an object-oriented programming concept where the internal state (data/fields) of an object is protected from direct, unauthorized modification by external code. It is the mechanism of restricting access to the data members and ensuring that they can only be manipulated through the object's defined methods.

How to Achieve It in Java:

You achieve data hiding in Java primarily using the `private` access modifier.

1. Declare Fields as `private`: Use the `private` keyword when declaring instance class variables.
2. **Provide Public Accessors/Mutators:** Offer public methods (**Getters** and **Setters**) to provide controlled read and write access to those private fields. This pattern is the foundation of **Encapsulation**.

2. Data Encapsulation

Encapsulation is the concept of **bundling data (fields) and the methods that operate on that data** into a single unit (a class).

- It is the overall principle that ties together **data hiding** (protecting the state) and **abstraction** (hiding complexity).
- **Benefit:** It allows the class to enforce business rules and maintain data integrity, as external code cannot directly manipulate the object's internal state in an uncontrolled way.

3. Data Abstraction

Abstraction is the process of hiding the complex implementation details and showing only the essential features of an object or system to the user.

- It focuses on what an object does rather than how it achieves it.
- In **Java**, abstraction is achieved using **Abstract Classes** and **Interfaces** (see OP). For example, when you use a method like `List.add(item)`, you only care about the result (the item is added), the underlying array resizing logic is abstracted away.

4. Static Polymorphism (Compile-Time)

Static Polymorphism, also known as **Compile-Time Polymorphism**, is achieved through **Method Overloading**.

- **Mechanism:** Defining multiple methods within the **same class** with the **same name** but with **different parameter lists** (number or type of arguments).
- **Resolution:** The compiler determines which method to call based on the arguments provided at **compile time**.

5. Runtime Polymorphism (Dynamic)

Runtime Polymorphism, also known as **Dynamic Polymorphism**, is achieved through **Method Overriding**.

- **Mechanism:** A subclass provides a specific implementation for a method that is already defined in its superclass.
- **Resolution:** The actual method to be executed is determined **at runtime** by the JVM, based on the **actual object type** (not the reference type). This requires an **"is-a"** relationship (Inheritance).

6. Inheritance

Inheritance is an object-oriented concept where a new class (**subclass** or **child class**) is derived from an existing class (**superclass** or **parent class**).

- **Relationship:** It establishes an **"is-a"** relationship (e.g., a "Car is a Vehicle").
- **Benefit:** It promotes **code reusability** because the subclass automatically inherits the fields and methods of the superclass.

7. Interface

An **Interface** in Java is a blueprint of a class. It can contain **method signatures** (abstract methods) and **constants**. Since Java 8 it can also contain **default** and **static** methods.

Benefits of Interface:

1. **Abstraction:** Hides implementation details completely.
2. **Achieve Total Decoupling:** Classes that implement an interface are only bound by the method contracts, making the code flexible.
3. **Support for Polymorphism:** Allows a reference variable of the interface type to hold objects of any class that implements the interface (runtime polymorphism).
4. **Simulate Multiple Inheritance of Type:** Java does not allow multiple class inheritance, but a class can implement multiple interfaces, allowing it to inherit contractual behavior from many sources.

8. Interface vs. Inheritance Choice

This choice is a fundamental design decision between establishing a hierarchy (Inheritance) or defining a "plug-in" contractual behavior (Interface). Inheritance is for genuine "is-a" relationships where code reusability of implementation is crucial.

- **Choose Interface (Preferable):** When defining a **contract** or a **capability**. For example, `Runnable`, `Serializable`, or `Printable`. This allows unrelated classes to share a common behavior without being tightly coupled in the same hierarchy. This is crucial for **loose coupling** and **flexibility**.
- **Choose Inheritance:** When one class is truly a **sub-type** of another, and you want to **reuse the actual implementation** (fields and methods) from the parent class.

9. Difference Between Abstract Class and Interface

Feature	Abstract Class	Interface
Type of Members	Can have **abstract** (no body) and **concrete** (with body) methods.	Can have **abstract** methods. Since Java 8 can also have **default** or **static** methods.
Fields	Can have any type of variables (instance, static, final).	Fields are implicitly `public static final` (constants).
Access Modifiers	Can have fields/methods with `public`, `protected`, `private`, or default access.	Members are implicitly `public`. Before Java 9).
Inheritance	A class can **extend** only **one** abstract class.	A class can **implement multiple** interfaces.
Constructor	Can have constructors (used by subclasses).	Cannot have constructors.
Purpose	To provide a partial implementation and share code among related classes.	To define a contract/type that unrelated classes can fulfill.

Export to Sheets

10. Static Members

Member	Definition	Use Scenario
Static Variable	A variable belonging to the class rather than to any instance. Initialized only once.	Used for constants (e.g., `PI`), counters (to track total instances), or shared resources.
Static Method	A method belonging to the class, not to any instance. Cannot access instance variables or methods directly.	Used for **utility functions** that don't rely on object state (e.g., `Math.sqrt()`). Also used for factory methods and helper methods. Backing variables.
Static Block	A block of code executed once when the class is loaded into the JVM memory.	Used for **initializing static variables** that require complex logic (e.g., reading configuration from a file, initializing a static collection).

Export to Sheets

11. The `final` Keyword

Use with	Restriction
Variable	The variable's value, once assigned, cannot be changed. It becomes a **constant**.
Method	The method cannot be overridden in a subclass.
Class	The class **cannot be subclassed** (inherited). Used for security and immutability (e.g., `String`).

Export to Sheets

12. Package In Java

A **package** is a mechanism to **group related classes and interfaces** together. It is a way to organize code, similar to folders in a file system.

Significance:

1. **Naming Conflict Prevention:** Packages create a **namespace**, preventing classes in different packages from having naming clashes (e.g., `java.util.List` is different from `java.awt.List`).
2. **Access Control:** Packages provide a level of access control (**package-private** or default access) which limits visibility to only classes within the same package.
3. **Maintainability:** Grouping related code makes large projects easier to manage, locate, and maintain.

13. Modifiers (Access Modifiers)

A **modifier** is a keyword used to alter the definition of a class, method, or variable. **Access Modifiers** specifically control the visibility (scope) of a class, method, or field.

Modifier	Accessible Within the...	Description
`private`	**Same Class** only.	Highest restriction. Hides implementation details.
(Default)	**Same Package** only.	No keyword is used. Often called "package private".
`protected`	**Same Package** AND **Subclasses** (in any package).	Used when you want sub-types to access a member, even if they're in a different package.
`public`	**Everywhere** (Classes, Packages, Subclasses, World).	Lowest restriction. Universal visibility.

Export to Sheets

14. `String` and `StringBuilder`

Feature	`String`	`StringBuilder`
Mutability	**Immutable** (Cannot be changed after creation).	**Mutable** (Can be changed after creation).
Memory	Each modification creates a **new** object in the String pool. Wasteful in the loop.	Modifications are performed on the **same** object in the heap.
Thread Safety	Naturally thread-safe due to immutability.	**Thread-safe** (Methods are synchronized).
Performance	Slower when performing many modifications (due to creating many objects).	Generally slower than `StringBuilder` in single-threaded environments due to synchronization overhead.

Export to Sheets

15. Immutable and Mutable Classes

Immutable Classes

- **Definition:** An object whose state **cannot be changed** after it is created.
- **Scenario:** Use immutable classes when you need objects to be **thread-safe** (as their state never changes) and for use as **keys** in a `HashMap`. (since the hash code depends on the state). `String` is the classic example.

Mutable Classes

- **Definition:** An object whose state **can be changed** after it is created.
- **Scenario:** Use mutable classes when you need to perform **frequent modifications** to an object's state, as this is much more efficient than creating a new object for every change. `StringBuilder` and `StringBuilder` are examples.

16. String Pool

The String Pool (or String Constant Pool) is a special storage area within the Heap memory for literals. Historically, the PermGen/Metaspace where `String` literal objects are stored.

- When you create a String using a literal (e.g., `String s = "hello";`), the JVM first checks the String Pool. If "hello" exists, the reference to the existing object is returned. If not, a new object is created and placed in the pool.
- **Creation using `new`:** Using `String s = new String("hello");` forces the creation of a new object on the regular Heap, even if "hello" exists in the pool.

Benefit of String Pool:

The main benefit is **Memory Optimization** and **Performance**. It ensures that identical string literals are represented by the **same object instance**, saving memory and allowing for faster string comparisons using the `==` operator for literals.

17. `StringBuffer` vs. `StringBuilder`

Feature	`StringBuffer`	`StringBuilder`
Thread Safety	Thread safe (Methods are synchronized).	Not Thread safe (Methods are not synchronized).
Performance	**Slower** in a single-threaded environment due to synchronization overhead.	**Faster** in single-threaded environment due to lack of synchronization overhead.
Use Case	Use when **multiple threads** might access the object to avoid race conditions concurrently.	Use in **single-threaded** environments or when thread-safety synchronization is handled.

Export to Sheets

To improve performance, you should use `StringBuilder` in a single-threaded environment because the absence of synchronization makes its operations faster.

18. Achieving Multiple Inheritance in Java

Java does **not** support multiple inheritance of **classes** (a class can only extend one superclass) to avoid the "Diamond Problem" (ambiguity of which superclass method to inherit).

However, Java achieves **multiple inheritance of type/contract** using **interfaces**.

- A class can implement multiple interfaces.

  Java

  ```
  class MyClass implements InterfaceA, InterfaceB {
      // Must implement all abstract methods from A and B
  }
  ```

- Since Java 8, interfaces can contain **default methods**, which provides a form of **implementation inheritance** from multiple sources, as long as there are no conflicting signatures.

19. The `@Override` Annotation

The `@Override` annotation is a **marker** that indicates a method is a subclass is intended to **override** a method in its superclass or implement a method from an interface.

Is It Mandatory?

No, it is not **mandatory** for the method to function as an override.

Why It is Better to Use `@Override` in Runtime Polymorphism:

It is strongly recommended because it provides a crucial **Compile-Time Check**:

- If you use `@Override`, but the method name, the parameter list, or the return type differ, the compiler will **immediately throw an error**.
- Without the annotation, a mistake would simply result in **method overloading** (creating a new, unrelated method), silently breaking the intended runtime polymorphism behavior, which is a difficult bug to find.

20. Constructors

A **Constructor** is a special type of method used to **initialize an object** when it is created using the `new` keyword.

Default Constructor

- Definition: A constructor that takes no arguments.
- **Automatic Creation:** If you do not define any constructor in a class, the Java compiler automatically inserts a **public no-argument constructor** (the default constructor).
- **Usage:** Used to create an object with all fields initialized to their default values.

Constructor with Arguments

- **Definition:** A constructor that accepts one or more parameters.
- **Usage:** Used to initialize the object's fields with specific values provided at the time of object creation, ensuring the object is created in a valid, ready-to-use state.

21. Significance of `super` Keyword

The `super` keyword is used to refer to the **immediate parent class object**.

Significance/Use Cases:

1. **Accessing Parent Class Members:** To explicitly access a field or method from the parent class when the subclass has a member with the same name (shadowing).
   - Calling a parent class overrideable/non-final primary key or a method specific implementation of this superclass from the subclass constructor using `superclass(parent)`. This must be the **first statement** in the subclass constructor.
2. **Preventing Ambiguity:** Resolving ambiguity when multiple methods or fields have the same name in the hierarchy.

22. Use of `this` Keyword

The `this` keyword is used to refer to the **current object instance**.

Use Cases:

1. **Referring to Current Instance Fields:** To distinguish between instance variables and local/parameter variables that have the same name (e.g., in a setter or constructor).

   Java

   ```
   public void setID(int id) {
       this.id = id; // "this.id" is the field, "id" is the parameter
   }
   ```

2. **Calling Current Class Constructor:** To call one constructor of the class from another constructor in the same class (Constructor Chaining), using `this(parameters)`. This must be the **first statement** in the calling constructor.
3. **Returning Current Instance:** To return the current object from a method (e.g., in method chaining).
4. **Passing Current Instance:** To pass the current object as an argument to another method.

1. What is exception handling.
2. What are the classes given by java in exception handling...

Here are the answers to your questions about Java exception handling.

1. What is Exception Handling?

Exception handling is a programming mechanism that manages and responds to **runtime errors** (exceptions) in a controlled manner, preventing the program from crashing and ensuring the application can continue execution or shut down gracefully. It separates the normal business logic from the error handling code.

2. Exception Handling Classes in Java

All exception and error classes in Java are derived from the parent class, `Throwable`. The main hierarchy is:

- `Java.lang.Throwable`: The base class for all exceptions and errors.
- `Java.lang.Exception`: The base class for exceptions that a typical application should catch and handle. These include checked and unchecked exceptions.
- `Java.lang.RuntimeException`: a subclass of `Exception`, serving as the base class for unchecked exceptions (programming errors).
- `Java.lang.Error`: The base class for serious, unrecoverable problems (e.g., system crashes) that errors generally are not caught via the application.

3. What does `try-catch-finally` do?

These are the keywords used for a standard exception handling block:

- `try` : Encloses the code segment that is expected to **potentially throw an exception**. If an exception occurs here, control is immediately transferred to the corresponding `catch` block.
- `catch` : Encloses the code that **handles the specific exception** declared in its argument. A `try` block can have multiple `catch` blocks to handle different types of exceptions.
- `finally` : Encloses code that **must be always executed**, regardless of whether an exception was thrown or caught. It is typically used for **cleanup activities** like closing files, network connections, or other external resources.

4. Difference between `final`, `finally`, and `finalize`

Keyword	Type	Scope/Purpose
`final`	**Modifier**	Used to make a variable a **constant**, a method unable to be **overridden**, or a class unable to be **subclassed**.
`finally`	**Block**	A block of code in exception handling that **always executes**, used for cleanup.
`finalize()`	**Method**	A deprecated method of the `Object` class that the **Garbage Collector** attempts to call on an object just before reclaiming its memory.

Export to Sheets

5. What is "Try-with-Resources"?

Try-with-Resources (introduced in Java 7) is an enhancement to the `try` block designed for **automatic resource management**.

- It declares one or more **resources** (objects that implement `java.lang.AutoCloseable`) inside parentheses following the `try` keyword.
- The resources are guaranteed to be **closed** automatically by the JVM when the block finishes either normally or abruptly (due to an exception). This eliminates the need for an explicit `finally` block to handle resource closing and prevent resource leaks.

Example:

```
try (Scanner scanner = new Scanner(new File("file.txt"))) {
    // use scanner
} // scanner is automatically closed here
```

6. Difference between Exception and Error

Feature	`Exception` (Subclass of `Throwable`)	`Error` (Subclass of `Throwable`)
Severity	Less severe. Typically, the application can recover.	Highly severe, critical, and generally unrecoverable.
Origin	Caused by application or user issues (e.g., file not found, bad input, rules out of bounds).	Caused by environmental issues in the JVM or system (e.g., resource exhaustion).
Handling	Can be caught and handled by the program.	Failure of the JVM itself.

Export to Sheets

Few Errors In Java:

1. `StackOverflowError` : Occurs when the application recurses too deeply (infinite recursion).
2. `OutOfMemoryError` : Occurs when the JVM runs out of heap memory.
3. `VirtualMachineError` : Indicates a serious problem with the JVM itself.

7. Topmost Class of Java for Exception

The topmost class in the entire Java exception and error hierarchy is `java.lang.Throwable`.

8. Difference between `Exception` and `RuntimeException`

Both are classes that extend `Throwable`, but they differ in their categories of exceptions.

Class	Type of Exception	Compiler Check?	Handling
`Exception` (excluding `RuntimeException`)	**Checked Exception**	**Yes** (Compiler forces handling/declaration).	Used for **predictable but recoverable** conditions (e.g., `IOException`, `SQLException`).
`RuntimeException`	**Unchecked Exception**	**No** (Compiler does not force handling/declaration).	Used for **programming errors** (e.g., null pointers, illegal arguments, array index out of bounds).

Export to Sheets

9. Difference between `throw` and `throws`

Feature	`throw`	`throws`
Usage	Used to **actually throw** an instance of an exception object at a method's body.	Used in the method signature to declare which exceptions a method might throw.
Followed by	An object instance of the `Throwable` class (e.g., `new MyException()`, `IOException(e)`).	Followed by a list of the `Throwable` class types (e.g., `IOException, SQLException`).
Quantity	Can only throw **one** exception at a time.	Can declare **multiple** exceptions, separated by commas.

Export to Sheets

10. `NullPointerException` and `throws`

No, you do not need to specify `NullPointerException` **or any of its subclasses (unchecked exceptions) using the** `throws` **keyword** in the function signature.

Why?

`NullPointerException` s are considered **unchecked** because they usually indicate flaws in the program's logic (programming errors) that cannot be reasonably predicted or recovered from at the point of call.

The compiler treats unchecked exceptions differently:

1. It assumes they can occur anywhere.
2. It does **not** force the programmer to either catch or declare them.
3. Declaring them with `throws` is **logically unnecessary** boilerplate, as the compiler ignores it for mandatory checking.

11. How to Create Custom Exception

To create a **custom exception** (also known as user-defined exception), you simply create a new class that **extends** an existing exception class, typically `Exception` or `RuntimeException`.

1. **Define the Class**: Create a class extending one of the base exception classes.
2. **Implement Constructors**: It's standard practice to provide constructors that call the superclass constructor, particularly those that accept a `String` message and a `Throwable` cause.

Example:

```
public class InsufficientFundsException extends Exception {
    // Constructor for a simple message
    public InsufficientFundsException(String message) {
        super(message);
    }
}
```

12. Custom Exception: Extending `Exception` vs. `RuntimeException`

The choice dictates whether your custom exception will be **checked** or **unchecked**.

Base Class	Custom Exception Type	Scenario for Use
`Exception`	**Checked Exception**	Use for **recoverable** conditions that a calling method should be aware of and handle (e.g., custom validations, account already exists, file access denied).
`RuntimeException`	**Unchecked Exception**	Use for **programming errors** or unrecoverable conditions (e.g., `IllegalArgumentException`, `NullPointerException`) where the error is propagated up the call stack and potentially crash the thread, indicating a bug that needs fixing.

Export to Sheets

13. What Kind of Code I Will Write in `catch` Block

The code in a `catch` block should focus on **handling the exception** and returning the application to a stable state. This includes:

- **Logging**: Recording the exception details (message, stack trace) for debugging purposes.
- **Displaying User-Friendly Messages**: Presenting an appropriate error message to the user instead of the technical stack trace.
- **Graceful Recovery**: Trying to fix the issue if possible (e.g., prompting the user for a different input, attempting an alternative connection).
- **Re-throwing**: If the current method cannot fully handle the exception, it might wrap it in a new exception and re-throw it (`throw new MyException(e)`) to propagate it further.
- **Exiting**: Shutting down the thread or the application if the error is critical.

14. Can I have Nested `try-catch` Inside a `try` Block?

Yes, you can have nested `try-catch` blocks.

- This is useful when you want to handle different types of exceptions at different levels of execution granularity.
- The inner `try` block handles its specific set of exceptions. If it fails to catch an exception, that exception is propagated outward to the outer `catch` blocks for processing.

15. Difference between Checked and Unchecked Exception

Feature	Checked Exception	Unchecked Exception
Inheritance	Directly or indirectly extends `Exception` but not `RuntimeException`.	Extends `RuntimeException` or `Error`.
Compiler Check	Compiler forces you to either handle it with `try/catch` or declare it with `throws`.	Compiler does NOT force handling or declaration.
Recoverability	Generally **recoverable** and anticipated (e.g., `IOException`).	Usually caused by **programming errors** and often unrecoverable (e.g., `NullPointerException`).

Export to Sheets

16. Examples of Checked and Unchecked Exceptions

Checked Exceptions (Must be handled/declared):

- `IOException` : General input/output operations.
- `FileNotFoundException` : When a file cannot be opened.
- `SQLException` : Database access errors.

Unchecked Exceptions (Do NOT need to be handled/declared):

- `NullPointerException` : Attempting to use a reference that has a null value.
- `ArrayIndexOutOfBoundsException` : Accessing an array out of bounds.
- `ClassCastException` : Attempting to cast an object to a type of which it is not an instance.
- `IllegalArgumentException` : A method has been passed an illegal or inappropriate argument.
- `ArithmeticException` : An exceptional arithmetic condition (e.g., division by zero).

> 1. How to make a class as Multithreading class.
> 2. When should you use Implements Runnable and when should you use extends Thread class.

Here are the answers to your questions on Java Multithreading and Concurrency.

1. How to Make a Class a Multithreading Class

To enable a class to be executed concurrently by a separate thread, it must define the code that the thread will run. This is done in Java by implementing the `Runnable` interface or extending the `Thread` class.

1. **Implement** `Runnable` : The class implements the `Runnable` interface and provides an implementation for the `run()` method.

```
class MyTask implements Runnable {
    public void run() {
        // Code to be executed by the thread
    }
    // ... other methods and fields ...
}
```

2. **Extend** `Thread` : The class extends the `Thread` class and overrides the `run()` method.

```
class MyThread extends Thread {
    public void run() {
        // Code to be executed by the thread
    }
}
// To execute: new MyThread().start();
```

2. **Implements** `Runnable` **vs.** `extends Thread`

Feature	`implements Runnable` (Recommended)	`extends Thread`
Inheritance	Allows the class to **extend another class**. Crucial for support multiple inheritance.	Consumes the single **inheritance slot**, preventing the class from extending any other class.
Data Sharing	Enables **easy data sharing** in multiple `Thread` objects are given the same `Runnable` object instance.	Each object is a unique `Thread`, making sharing slightly more complex (requires static members or careful passing).
Decoupling	**Separates the task (Runnable) from the execution mechanism (Thread)**, promoting better design.	Tightly couples the task and the execution mechanism.

Export to Sheets

General Rule: Always prefer `implements Runnable` unless you specifically need to override or modify core `Thread` behaviors.

3. Specify the Methods of the `Thread` Class

The `java.lang.Thread` class provides many methods for managing and controlling threads. Key methods include:

Method	Purpose
`void start()`	Starts the execution of the thread by calling its `run()` method. The JVM schedules it.
`void run()`	Contains the code executed by the thread. Must be overridden.
`static void sleep(long millis)`	Causes the currently executing thread to cease execution for a specified period.
`void join()`	Waits for this thread to die (complete its execution).
`void setPriority(int newPriority)`	Changes the priority of the thread.
`String getName()`	Returns the name of the thread.
`void setPriority(int newPriority)`	Changes the priority of the thread.
`static Thread currentThread()`	Returns a reference to the currently executing thread object.
`boolean isAlive()`	Tests if the thread is alive (has been started and has not yet died).
`void interrupt()`	Interrupts this thread.

Export to Sheets

4. Thread Priority

Thread Priority is a hint to the thread scheduler about the relative importance of a thread. Threads with higher priority are given preference in terms of execution time, though the actual scheduling is **highly dependent on the underlying OS and JVM implementation**.

Constants of Priority (Constants in `Thread` class):

Java thread priorities are integers ranging from 1 to 10.

1. `Thread.MIN_PRIORITY` : The minimum priority (Value **1**).
2. `Thread.NORM_PRIORITY` : The default priority (Value **5**).
3. `Thread.MAX_PRIORITY` : The maximum priority (Value **10**).

5. How to Set the Name for a Thread

Here are two primary ways to set the name of a thread:

1. **Using the** `Thread` **constructor**: Pass the desired name when creating the thread object.

```
Thread t1 = new Thread("MyNewThread");
// For Runnable: Thread t2 = new Thread(myRunnable, "UserThread-2");
```

2. **Using the** `setName()` **method**: Call the setter method on the `Thread` object after it has been created.

```
Thread t3 = new Thread();
t3.setName("ProcessorThread-3");
```

6. How to Synchronize a Method

To synchronize a method, you can use the `synchronized` keyword in the method declaration.

- **Purpose**: This ensures that **only one thread** can execute this method on the **same object instance** at any given time.
- **Mechanism**: When a thread enters a synchronized instance method, it automatically acquires the **intrinsic lock (monitor)** of the **object**. The lock is released when the thread exits the method.

```
public class Counter {
    private int count = 0;

    // Synchronized instance method
    public synchronized void increment() {
        count++;
    }

    // Synchronized static method locks the lock on the Class object
    public static synchronized void staticIncrement() {
        // ...
    }
}
```

7. Synchronized Block vs. Synchronized Method

Feature	Synchronized Method	Synchronized Block
Syntax	Keyword added to the method signature (`public synchronized void method()`).	Explicitly defined block (`synchronized (object) { ... }`).
Scope of Lock	The **entire method** is protected.	Only the code **inside the block** is protected.
Object Used	`this` (the current object) is locked for instance methods. The `Class` object is locked for static methods.	Locks the **explicit object reference** provided in parentheses.
Granularity	**Coarse grained**. Threads more could than necessary, potentially reducing concurrency.	**Fine grained**. Allows you to protect only the critical section of the code, improving concurrency.

Export to Sheets

8. What is `Thread.sleep()`

The static method `Thread.sleep(long millisecond)` causes the thread that is currently executing to pause its execution for at least the specified number of milliseconds.

- **State Change**: The thread moves from the **Running** state to the **Timed Waiting** state.
- **Lock Status**: The sleeping thread **retains** any monitor locks (synchronization locks) it may hold; it does not release them.
- **Interruption**: The sleep can be interrupted by another thread, which throws an `InterruptedException`.

9. What is the Purpose of `wait()`

The `wait()` method (from the `Object` class) is used for **inter-thread communication** to implement cooperation between threads (e.g., Producer-Consumer).

- **Purpose**: To pause the current thread's execution and release the intrinsic lock (monitor) it holds on the object.
- **Prerequisite**: Must be called from within a **synchronized context** (method or block).
- **State Change**: The thread moves from the **Running** state to the **Waiting** state and waits until another thread calls `notify()` or `notifyAll()` on the **same object**.

10. What is `notify()` and `notifyAll()`

`notify()` and `notifyAll()` are methods (from the `Object` class) used for waking up threads:

- `notify()` : Wakes up a **single thread** that is waiting on this object's monitor. If multiple threads are waiting, the choice of which one to wake up is arbitrary (JVM implementation dependent).
- `notifyAll()` : Wakes up **all threads** that are waiting on this object's monitor. All awakened threads then compete for the object's lock, which is only released after the thread that called `notifyAll()` exits its synchronized block.

Crucial Rule: Like `wait()`, both methods must be called from within a synchronized context and operate on the same object that the waiting threads are locked on.

11. How to Resume a Thread on Hold from `wait()`

A thread that is paused due to calling `wait()` can be resumed in two ways:

1. **Signal from another thread**: Another thread calls `notify()` or `notifyAll()` on the same object on which the thread is waiting.
2. **Timeout**: If the thread called the timed version of the wait (e.g., `wait(long timeout)`), it will automatically move out of the waiting state after the specified time elapses.

Steps after being awakened: After being notified, the thread moves from the **Waiting** state to the **Blocked/Monitor** state and must re-acquire the object's intrinsic lock before it can move back to the **Runnable** state and resume execution.

12. What Is Inter-Thread Communication

Inter-thread communication (ITC) is the process by which multiple threads coordinate their activities. It allows threads to share information and synchronize their actions to avoid race conditions, ensuring that threads execute in a specific order and don't interfere with each other's work (e.g., waiting for data to become available).

13. How to Achieve Inter-Thread Communication

Inter-thread communication in Java is primarily achieved through the methods of the `Object` class, which operate on the **object's monitor (intrinsic lock)**:

1. `synchronized`: Ensures exclusive access to the shared resource.
2. `wait()`: Causes a thread to release the lock and wait for a notification.
3. `notify()` / `notifyAll()`: Wake up waiting threads.

This mechanism is often used to implement the **Producer-Consumer pattern**.

14. Program to Perform Inter-Thread Communication

Here is the classic Producer-Consumer example using `wait()` and `notify()` to synchronize the production and consumption of a single item in a shared memory pool.

Java

```
class DataStore {
    private String data;
    private boolean available = false;

    public synchronized void produce(String newData) throws InterruptedException {
        while (available) { // While data is available, producer must wait
            wait();
        }
        this.data = newData;
        System.out.println("Produced: " + newData);
        available = true;
        notify(); // Notify consumer that data is ready
    }

    public synchronized String consume() throws InterruptedException {
        while (!available) { // While data is NOT available, consumer must wait
            wait();
        }
        String consumedData = this.data;
        System.out.println("Consumed: " + consumedData);
        notify(); // Notify producer that it can produce again
        return consumedData;
    }
}
```

15. How to Share Data Between Threads

Data can be shared between threads in Java primarily through a **shared object** instance:

1. **Shared `Runnable` Object:** When using `implements Runnable`, create a single instance of the class and pass it to multiple `Thread` objects. All threads operate on the same instance's fields.
2. **Static Fields:** Threads can share data via **static fields** of a class. Access to these fields must be synchronized.
3. **Concurrent Collections:** Use thread-safe data structures from the `java.util.concurrent` package (e.g., `ConcurrentHashMap`, `LinkedBlockingQueue`) which manage internal synchronization, simplifying the code.
4. `ThreadLocal`: Provides thread-local variables (like `ThreadLocal`) which are inherently thread-safe since their state cannot change.

16. Difference between a Thread and a Process

Feature	Process	Thread
Definition	An independent program unit with its own memory space.	A single sequence of execution within a process.
Memory	Each process has its **own dedicated address space** (heap, stack, data).	Threads **share the process's memory** (heap, data) but have their own stack and PC.
Communication	Communication between processes is slower and uses IPC mechanisms (e.g., pipes, sockets).	Communication threads is faster and easier due to shared memory.
Context Switch	Slower, as it involves changing the memory map and kernel data.	Faster, as it only involves saving/restoring the thread's registers and stack pointer.

Export to Sheets

17. Difference between `start()` and `run()`

Feature	`start()`	`run()`
Execution	Creates a **new thread of execution** and calls the `run()` method on that thread.	Executes the thread code **sequentially** within the **current** thread (like a regular method call).
Multithreading	Enables multithreading. Execution is asynchronous.	**Does NOT enable** multithreading. Execution is synchronous.
Call Frequency	Called **only once** on a thread object. Calling it twice throws an `IllegalThreadStateException`.	Can be called **multiple times** just like any other method.

Export to Sheets

In summary: You call `start()`, and the JVM calls `run()` on the new thread.

18. Thread Life Cycle

A thread typically goes through five major states, often referred to as the **Thread Life Cycle**:

1. **`New`:** The thread has been created but has not yet started (i.e., `new Thread()` has been called, but `start()` has not).
   - **Function:** None specific in this state; object creation only.
2. **`Runnable` (Ready):** The thread is waiting for the CPU and is ready to run. It could be running or waiting its turn.
   - **Function:** `start()` moves the thread to this state.
3. **`Running`:** The thread is currently executing on the CPU.
   - **Function:** handled by the **Thread Scheduler** selecting a thread from the Runnable pool.
4. **`Blocked/Waiting/Timed Waiting`:** The thread is temporarily inactive and cannot run.
   - **Functions:**
     - `sleep()` — Thread Waiting (waiting task)
     - `join()` — Waiting (waiting task)
     - `wait()` — Waiting (waiting task)
     - **Waiting for Lock/IO** — Blocked (waiting for IO completion or lock acquisition).
5. **`Terminated (Dead)`:** The thread has completed its execution (`run()` method has finished) or has exited due to an unhandled exception.
   - **Function:** The `run()` method completes naturally, or an unhandled exception occurs.

19. Race Condition and Prevention

A **Race Condition** occurs when two or more threads access shared data concurrently and try to modify it, and the outcome of the computation depends on the unpredictable order in which the threads access the data. This leads to inconsistent and often incorrect results.

How to Prevent It:

Race conditions are prevented by ensuring that only one thread can execute the **critical section** (the code accessing the shared data) at a time. This is known as mutual exclusion:

1. **Synchronization:** Use the `synchronized` keyword on methods or code blocks to acquire the object's intrinsic lock.
2. **Locks:** Use explicit locking mechanisms from the `java.util.concurrent.locks` package (e.g., `ReentrantLock`).
3. **Atomic Variables:** Use classes from `java.util.concurrent.atomic` (e.g., `AtomicInteger`) for single-variable operations, which use non-blocking machine instructions (CAS - Compare And Swap).
4. **Concurrent Collections:** Use thread-safe collections (e.g., `ConcurrentHashMap`).

20. What is a Thread Pool

A **Thread Pool** is a managed collection of worker threads that are kept alive to execute incoming tasks. Instead of creating a new thread for every task, the tasks are submitted to the pool, and the threads in the pool execute them.

Benefits:

- **Reduced Overhead:** Eliminates the performance overhead of constantly creating and destroying threads.
- **Resource Management:** Limits the number of threads running concurrently, preventing resource exhaustion (e.g., excessive memory use or context switching).
- **Improved Responsiveness:** Tasks can be started almost immediately as threads already exist.

How to Create It in Java:

The most common way to create and manage a thread pool in Java is by using the `ExecutorService` framework from the `java.util.concurrent` package, specifically via the static factory methods of the `Executors` class:

Java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

// 1. Fixed-Size Thread Pool (Most common)
ExecutorService fixedPool = Executors.newFixedThreadPool(10);

// 2. Single Thread Executor (Tasks run sequentially)
ExecutorService singleExec = Executors.newSingleThreadExecutor();

// 3. Cached Thread Pool (Creates new threads as needed, reuses idle ones)
ExecutorService cachedPool = Executors.newCachedThreadPool();

// Submit a task to the pool
fixedPool.submit(new Runnable() {
    public void run() {
        System.out.println("Task executed by pool thread.");
    }
});

// IMPORTANT: Shut down the pool when done
fixedPool.shutdown();
```

> 1. What is functional interface. Why do we need it. What `@FunctionalInterface` can do for us. How do we specify it in Java.
> 2. Write a Lambda function and forEach on array.

Here are the answers to your questions on Java 8 features (Functional Interfaces, Lambda Expressions, and Streams).

1. Functional Interface

A **Functional Interface** is an interface that contains **exactly one abstract method**.

- **Why we need it:** Functional Interfaces serve as the **target type** for **lambda expressions** and **method references**. They provide the necessary context (the single abstract method signature) that the lambda expression's implementation fulfills.
- **Benefit:** They enable **functional programming** paradigms in Java, allowing you to pass behavior (code) as an argument to methods, which is crucial for the Streams API and simplified concurrency.
- **How to Specify it:** You specify a functional interface using the `@FunctionalInterface` annotation. While optional, it is highly recommended because it forces the compiler to check that the interface adheres to the single abstract method rule.

Java

```
@FunctionalInterface
interface MyMessage {
    void sayHello(String s);
    // Can also contain default and static methods (Java 8+)
}
```

2. Lambda Function (Lambda Expression)

A **lambda expression** is a concise way to represent an **anonymous function** (a function without a name). It provides a compact syntax to implement the single abstract method of a functional interface.

- **Syntax:** `(parameters) -> { body }` or `(parameters) -> expression`
- **Benefit:**
  - **Code Conciseness:** Eliminates the need for verbose anonymous inner classes.
  - **Readability:** Makes code that operates on collections (like streams) much clearer.
  - **Functional Programming:** Allows behavior to be treated as data, making it easy to pass blocks of code to methods.

Java

```
// Traditional implementation using an anonymous class
// new ComparatorString() { public int compare(String s, String b) { return s.le

// Lambda implementation
(s, b) -> s.length()
```

3. Method References

Method References are compact, readable syntax forms used to refer to a method without executing it. They are a special type of lambda expression that provides an even simpler syntax for methods that already have a name.

- **Syntax:** `ClassName::methodName`
- **Different Kinds:**
  1. **Reference to a Static Method:** `ClassName::staticMethodName`
  2. **Reference to an Instance Method of a Particular Object:** `object::instanceMethodName`
  3. **Reference to an Instance Method of an Arbitrary Object of a Particular Type:** `ClassName::instanceMethodName`
  4. **Reference to a Constructor:** `ClassName::new`

Type	Example	Usage
Static	`Math::max`	`(x, b) -> Math.max(x, b)`
Instance (Object)	`myPrinter::println`	`(message) -> myPrinter.println(message)`
Arbitrary	`String::length`	`(str) -> str.length()`
Constructor	`ArrayList::new`	`() -> new ArrayList()`

Export to Sheets

4. `Optional` Class in Java

The `Optional<T>` (`java.util.Optional`) class is a container object used to represent the **presence or absence of a value**.

- **Purpose:** Its primary purpose is to provide a clear, declarative alternative to using `null` references, which prevents common errors like `NullPointerException` (NPE).
- **Benefit:** It forces developers to explicitly check for the presence of a value, making code more robust and readable by clearly indicating that a value might be missing.

Java

```
Optional<String> optional = Optional.ofNullable(getMyString());

// Safe way to get the value or a default
String result = optional.orElse("Default value");

// Execute an action only if value is present
optional.ifPresent(s -> System.out.println("Value is " + s));
```

5. Default Methods in Interfaces

Default methods (or defender methods) are methods defined in an interface with the `default` keyword that have a **concrete implementation (method body)**.

- **Purpose:** They were introduced in Java 8 to allow new methods to be added to an interface **without breaking existing classes** that implement that interface.
- **Behavior:** Any class implementing the interface automatically inherits the default implementation. The implementing class can optionally override the default method.
- **Significance:** This was essential for adding new functionality like `forEach()` to interfaces like `List` and `Collection`.

Java

```
interface Vehicle {
    void start();// Abstract method

    default void printInfo() {
        System.out.println("It is running.");// Default implementation
    }
}
```

6. Streams

(a) What are the Streams?

Streams are sequences of elements that support **sequential and parallel aggregate operations**. A Stream is **not** a data structure. It's a way to process data from a source (like a collection, array, or IO resource) in a functional and declarative manner.

(b) What is `parallelStream()`?

`parallelStream()` is a method that returns a parallel implementation of the Stream.

Feature	`stream()` (Sequential Stream)	`parallelStream()` (Parallel Stream)
Execution	Processing happens sequentially on a **single thread**.	Processing happens concurrently by splitting the work across **multiple threads** using the underlying Fork/Join framework (common pool).
Performance	Better for **small** data sets or operations that are IO-bound.	Better for **large** data sets and CPU-intensive operations.
Order Guarantee	Always maintains the order of elements (if source is ordered).	Order is generally maintained, but the internal execution is non-deterministic.

Export to Sheets

(c) Intermediate Functions of Stream

Intermediate operations transform a stream into another stream. They are **lazy**: they are not executed until a terminal operation is called.

Function	Explanation	Code Example
`filter(Predicate)`	Selects elements matching a given condition.	`list.stream().filter(n -> n > 0)`
`map(Function)`	Transforms each element into a new value.	`list.stream().map(s -> s.toUpperCase())`
`flatMap(Function)`	Transforms each element into a stream, then flattens these into a single stream.	`list.stream().flatMap(s -> Arrays.stream(s.split(",")))`
`distinct()`	Removes duplicate elements (uses `equals()`).	`list.stream().distinct()`
`sorted()`	Sorts the stream using natural order or a custom `Comparator`.	`list.stream().sorted()`
`peek(Consumer)`	Performs an action on each element as it passes through the stream (useful for debugging).	`list.stream().peek(System.out::println)`
`limit(long maxSize)`	Truncates the stream to contain only the first `maxSize` elements.	`list.stream().limit(10)`
`skip(long n)`	Skips the first `n` elements of the stream.	`list.stream().skip(5)`
`takeWhile(Predicate)`	Takes elements while the predicate is true (Java 9+).	`list.stream().takeWhile(n -> n < 10)`
`dropWhile(Predicate)`	Drops elements while the predicate is true (Java 9+).	`list.stream().dropWhile(n -> n < 5)`

Export to Sheets

(d) Terminal End functions

Terminal operations initiate the processing of the stream pipeline and produce a final result (a non-stream value or a side effect). After a terminal operation, the stream is considered **consumed** and cannot be reused.

Function	Explanation	Code Example
`forEach(Consumer)`	Performs an action for each element.	`list.stream().forEach(System.out::println)`
`collect(Collector)`	Accumulates elements into a mutable result container (e.g., List, Map, Set).	`list.stream().collect(Collectors.toList())`
`reduce(Binary<operator>)`	Combines elements into a single result using a binary operator or a reduction function.	`list.stream().reduce(0, (a, b) -> a + b)`
`count()`	Returns the number of elements in the stream.	`list.stream().count()`
`min(Comparator)`	Returns an `Optional` describing the minimum element.	`list.stream().min(Comparator.naturalOrder())`
`max(Comparator)`	Returns an `Optional` describing the maximum element.	`list.stream().max(Comparator.naturalOrder())`
`anyMatch(Predicate)`	Returns `true` if any element matches the provided predicate.	`list.stream().anyMatch(n -> n > 100)`
`allMatch(Predicate)`	Returns `true` if all elements match the provided predicate.	`list.stream().allMatch(n -> n > 0)`
`noneMatch(Predicate)`	Returns `true` if no elements match the provided predicate.	`list.stream().noneMatch(n -> n < 0)`

`flatMapInt()`	Returns an `IntStream` consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.	`list.stream().flatMapInt(s->{...});`
`flatMap()`	Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.	`list.stream().flatMap(s->{...});`
`toArray()`	Returns an array containing the elements of the stream.	`list.stream().toArray();`

Export to Sheets

(e) How to Sort an Attribute of an Object in Group of Records

You use the `sorted()` intermediate operation with `Comparator.comparing()` to specify the attribute to sort by:

```
// Sorting students by their total marks in descending order
List<Student> sortedStudents = students.stream()
    .sorted(Comparator.comparingInt(Student::getTotalMarks).reversed())
    .collect(Collectors.toList());

// Sorting students by name in descending order
List<Student> sortedByNameDesc = students.stream()
    .sorted(Comparator.comparing(Student::getName).reversed())
    .collect(Collectors.toList());
```

(f) Filter, Sort, Map an Attribute

```
// Filtering students with total marks greater than 500, sorting by name, and mapping to their names
List<String> filteredSortedNames = students.stream()
    .filter(s -> s.getTotalMarks() > 500)
    .sorted(Comparator.comparing(Student::getName))
    .map(Student::getName)
    .collect(Collectors.toList());
```

(g) Operations for Distinct, Sum, Avg, Count, Min, Max, and Reduce

Assuming a list of `Student` objects:

Operation	Code
Distinct	`students.stream().map(Student::getGrade).distinct().collect(Collectors.toSet());`
Count	`students.stream().count();`
Sum	`students.stream().mapToInt(Student::getTotalMarks).sum();`
Avg	`students.stream().mapToInt(Student::getTotalMarks).average();`
Min	`students.stream().min(Comparator.comparingInt(Student::getMarks));`
Max	`students.stream().max(Comparator.comparingInt(Student::getMarks));`
Reduce (Total Marks)	`students.stream().mapToInt(Student::getMarks).reduce(0, Integer::sum);`

Export to Sheets

(h) `filter()`, `flatMap()`, `flatMapInt()`, `limit()`, `skip()`

Assuming the `Student` class from above and `List<Student> students`:

Function	Code	Explanation
`filter()`	`students.stream().filter(s -> s.getMarks() > 80).forEach(...);`	Gets the students with marks > 80.
`flatMap()`	(Not a direct function) Achieved via `students.stream().map(s->{...}).flatMap(s -> s.||s||)` or mapping stream	Gets the list element in the stream
`flatMapInt()`	`students.parallelStream().flatMapInt(...);`	Gets any numbers from the Student in parallel stream.
`limit()`	`students.stream().limit(3).collect(Collectors.toList());`	Gets the first 3 students.
`skip()`	`students.stream().skip(2).collect(Collectors.toList());`	Skips the first 2 students, getting the rest.

Export to Sheets

Which one is better and why?

- **Choose** `forEach()` **(functional):** For most processing tasks where you simply need to apply an action to every element, `forEach()` is generally **better**. It is more concise, promotes a functional style, and is optimized by the JVM.
- Choose Enhanced `for` Loop (Traditional): When you need control flow, specifically the ability to use `break` to stop iterating early or `continue` to skip an element, the traditional `for` loop is the only choice.

> 1. What is IoC container
> 2. What is dependency injection
> 3. What is Setter based, constructor based and...

Here are the answers to your questions about Spring Core concepts.

1. What is IoC Container?

The IoC (**Inversion of Control**) Container is the core component of the Spring Framework. Its main responsibility is to **manage the lifecycle of Java objects** (beans). From creation to configuration and destruction. Instead of a developer managing object creation and dependencies manually, the container handles this automatically thereby inverting control of object creation and dependency management. The IoC container is typically implemented by `BeanFactory` and `ApplicationContext`. (the latter being the more advanced and commonly used one).

2. What is Dependency Injection (DI)?

Dependency Injection (DI) is a fundamental design pattern and the central mechanism by which the Spring IoC container achieves Inversion of Control.

Instead of an object creating or looking up its **dependencies** (other objects it needs to function), the dependencies are **injected** into it by an external entity (the IoC Container). This makes components loosely coupled, easier to test, and more reusable.

3. Setter-Based, Constructor-Based, and `@Autowired`

These refer to the three primary ways the Spring container performs Dependency Injection:

1. **Constructor-Based Injection**: Dependencies are provided as **arguments to the class** constructor. This is generally preferred as it ensures that the object is created in a valid state with its mandatory dependencies. Dependencies must be passed at creation time, making the class **immutable** if fields are `final`.
2. **Setter-Based Injection**: Dependencies are provided through **setter methods** after the object has been created by a no-argument constructor. This is useful for optional dependencies or for properties that may change.
3. `@Autowired`: This annotation, placed on a constructor, setter, or field, tells Spring to **automatically resolve and inject** the dependency at runtime.
   - **Field Injection (Least Preferred):** `@Autowired` placed directly on a field.
   - **Setter Injection:** `@Autowired` placed on the setter method.
   - **Constructor Injection (Best Practice):** `@Autowired` placed on the constructor (optional since Spring 4.3 if there's only one constructor).

4. What is Cyclic Dependency?

A **Cyclic Dependency** occurs when two or more beans depend on each other in a circular manner.

- Example: Bean A requires Bean B and Bean B requires Bean A.
- **Resolution:** Spring generally handles cyclic dependencies for **setter injection** (by creating one object and injecting the reference later). However, when using the preferred **constructor injection**, Spring typically throws a `BeanCurrentlyInCreationException` because it cannot complete the construction of either bean.

5. What is `@Qualifier`?

The `@Qualifier` (Qualifier)annotation is used in conjunction with `@Autowired` to **resolve ambiguity** when multiple beans of the same type are present. It specifies which bean to inject.

- When Spring finds two or more matching beans for a dependency, it throws a "NoUniqueBeanDefinitionException."
- **Usage:** You use `@Qualifier("beanName")` to specify exactly which bean to inject.

```
@Autowired
@Qualifier("emailService") // Tells Spring to inject the bean named "emailService"
private MessageService service;
```

6. What is `@Value`?

The `@Value` annotation is used to inject external values into fields in Spring components. These values typically come from:

1. **Property Files** (e.g., `application.properties`)
2. **Environment Variables**
3. **System Properties**
4. **SpEL (Spring Expression Language)**

```
@Value("${app.title}") // Injects the value of the "app.title" property
private String appTitleFromFile;

@Value("#{10 * 20}") // Injects the result of the SpEL expression (200)
private int calculatedValue;
```

7. 20 Spring Core Annotations

Category	Annotation	Purpose
Component Scanning	`@Component`	Generic stereotype for any Spring-managed component.
	`@Service`	Stereotype for business logic components (service layer).
	`@Repository`	Stereotype for data access/persistence components (DAO layer).
	`@Controller`	Stereotype for presentation layer components (MVC controllers).
	`@RestController`	Specialized controller for RESTful web services (combines `@Controller` and `@ResponseBody`).
Configuration	`@Configuration`	Indicates a class contains bean definition methods.
	`@Bean`	Marks a method that produces a bean to be managed by the Spring container.
	`@PropertySource`	Specifies the location of property files to load configuration from.
Dependency Injection	`@Autowired`	Marks a field, constructor, or setter for dependency injection.
	`@Qualifier`	Used with `@Autowired` to specify the exact bean name when multiple exist.
	`@Primary`	Designates a preferred bean when multiple matching types exist.
	`@Value`	Injects external values (properties, SpEL) into fields.
Lifecycle/Scope	`@Scope`	Defines the scope of the bean (e.g., singleton, prototype).
	`@Lazy`	Tells Spring to initialize the bean only when it is first requested, not at startup.
	`@DependsOn`	Forces Spring to initialize one or more specific beans before this bean initialization.
	`@PostConstruct`	Marks a method to be executed immediately after dependency injection is performed.
	`@PreDestroy`	Marks a method to be executed before the bean is destroyed by the container.
Modularity/Grouping	`@Import`	Imports one or more additional `@Configuration` classes.
	`@ImportResource`	Loads bean definitions from XML configuration files.
Environment	`@Profile`	Specifies that a component is only registered when a given profile is active (e.g., "dev", "prod").

Export to Sheets

8. What is the Scope of a Bean?

The **scope of a bean** defines the number of instances of that bean created and managed by the Spring container. The most common scopes are:

Scope	Description
`singleton` (default)	One single instance of the bean is created per Spring IoC container. This is the most common and efficient scope.
`prototype`	A **new bean instance** is created every time it is requested from the container.
`request`	A new bean instance is created for each **HTTP request** (Web-aware scope).
`session`	A new bean instance is created for each **HTTP session** (Web-aware scope).
`application`	A single bean instance is created for the **ServletContext** (Web-aware scope).

Export to Sheets

You define the scope using the `@Scope` annotation (e.g., `@Scope("prototype")`).

9. What is the Life Cycle of a Bean?

The **life cycle of a Spring bean** is the sequence of steps the bean IoC container takes from the time a bean is instantiated until it is destroyed.

Key Steps in the Bean Life Cycle:

1. **Instantiation:** The container creates the bean instance using its constructor.
2. **Populate Properties (Dependency Injection):** The container sets the bean's properties and injects dependencies (field, setter, or constructor injection).
3. **Bean Name/Factory/Context Awareness (if implementing `Aware` interfaces, e.g., `BeanNameAware`, `ApplicationContextAware`):**
4. **Post-Processing (Pre-Initialization):** Spring calls the `postProcessBeforeInitialization()` method of any registered `BeanPostProcessor` s.
5. **Initialization:**
   - If defined, the bean's custom `@PostConstruct` method (JSR-250).
   - If defined, the bean's custom `afterPropertiesSet()` method (`InitializingBean` interface).
   - If defined, the bean's custom `init-method` (XML configuration) is called.
6. **Post-Processing (Post-Initialization):** Spring calls the `postProcessAfterInitialization()` method of any registered `BeanPostProcessor` s.
7. **Ready for Use:** The bean is now ready for use by the application.
8. `BeanFactory`: When the container shuts down, or the scope ends:
   - If defined, the bean's custom `@PreDestroy` method (JSR-250).
   - If defined, the bean's custom `destroy()` method (`DisposableBean` interface) is called.
   - If defined, the bean's custom `destroy-method` (XML configuration) is called.

10. What is `@Autowired` and Different Types of `@Autowired`?

`@Autowired` is a core Spring annotation used to perform **automatic dependency injection**. It instructs the Spring container to automatically resolve and inject the required collaborator object into the annotated component.

Different Types of `@Autowired` **Injection:**

`@Autowired` can be placed in three different locations to achieve injection:

1. **Constructor Injection:** Placed on the constructor. Spring resolves all constructor arguments as dependencies and injects them. This **is the recommended approach** as it guarantees dependency availability and facilitates immutability.

```
@Autowired // Optional since Spring 4.3 for single constructor
public UserService(UserRepository repository) {
    this.repository = repository;
}
```

2. **Setter Injection:** Placed on a public setter method. Spring calls this method with the dependency after the bean has been instantiated.

```
@Autowired
public void setRepository(UserRepository repository) {
    this.repository = repository;
}
```

3. **Field Injection:** Placed directly on the field itself. Spring uses reflection to inject the dependency directly into the private field. This is generally discouraged because it makes testing difficult (it is impossible without the Spring container)and is not best and hides the dependencies.

```
@Autowired
private UserRepository repository;
```

> Spring MVC
> 1. What is the architecture of Spring MVC with explanation about each module.

Here are the answers to your questions about Spring MVC and Spring REST.

Spring MVC Architecture

The **Spring MVC (Model View Controller) architecture** is a request-driven framework designed around a central servlet, the `DispatcherServlet`. It implements the **Front Controller** pattern.

Architecture Modules and Explanation:

1. **Client (Browser):** Sends an HTTP request to the Spring MVC application.
2. **`DispatcherServlet` (Front Controller):** The central component. It intercepts every incoming request and delegates it to the appropriate handler. It is the entry point to the application.
3. **Handler Mapping:** Consults the available mappings (often defined by `@RequestMapping`) to find the **Controller** that can handle the request URL. It returns the corresponding handler (ControllerModel).
4. **Controller (Handler):** The request handler. It processes the user request by interacting with the **Service Layer** (business logic). It prepares the **Model** (data) and returns a logical **View Name**.
5. **Model:** A map of data (key-value pairs) prepared by the Controller, which is passed to the View to render it.
6. **View Resolver:** Takes the logical **View Name** returned by the Controller and maps it to an actual View implementation (e.g., a JSP file, Thymeleaf template, etc.).
7. **View:** The presentation component. It takes the **Model** data and renders the final response (e.g., HTML page) to be sent back to the client.
8. **Client (Browser):** Receives the rendered response (e.g., the HTML page).

Spring REST

1. What is Spring REST?

Spring REST or developing RESTful services with Spring refers to using the Spring framework, typically Spring Boot, to build **RESTful Web Services**. These services are stateless, follow the principles of the REST architectural style, and communicate primarily using standard **HTTP methods** and **JSON/XML** data formats.

2. Difference Between Spring REST and Spring MVC

Feature	Spring REST	Spring MVC
Primary Focus	Building traditional web applications that render **UI/HTML** pages.	Building APIs for client applications (mobile, frontend, third-party APIs).
Response Type	Primarily returns a **View Name** that maps to a template (HTML/JSP).	Primarily returns **Data** directly (usually **JSON** or **XML**).
Key Annotations	Uses `@Controller` and `@ResponseBody` (or `@RestController` which combines both).	Uses `@RestController` (which combines `@Controller` and `@ResponseBody`).
Client Type	Browser (coupled with the UI).	Any client capable of making HTTP requests (decoupled).

Export to Sheets

3. Spring REST Annotations

Annotation	Purpose	Example Usage
`@RestController`	A convenience annotation that combines `@Controller` and `@ResponseBody`. It marks a class as a handler for REST endpoints, implying all HTTP methods return data directly.	`public class UserController {}`
`@Request Mapping`	Maps HTTP requests to handler methods. Can specify the path, but also for all HTTP methods and paths.	`@Request Mapping("/api/v1/users")`
`@Get Mapping`	Specialized version of `@RequestMapping(method = GET)`.	`@GetMapping("/{id}")`
`@Post Mapping`	Specialized version of `@RequestMapping(method = POST)`.	`@PostMapping("/")`
`@Put Mapping`	Specialized version of `@RequestMapping(method = PUT)`.	`@PutMapping("/{id}")`
`@Delete Mapping`	Specialized version of `@RequestMapping(method = DELETE)`.	`@DeleteMapping("/{id}")`
`@Path Variable`	Binds a method parameter to a URI template variable (path segment).	`public User getUser(@PathVariable Long id)`
`@Response Status`	Marks a method or an exception class with a status code that should be returned.	`@ResponseStatus(HttpStatus.CREATED)`
`@Cross Origin`	Enables **Cross-Origin Resource Sharing (CORS)** globally or for specific handler classes/methods.	`@CrossOrigin(origins = "http://localhost:3000")`

Export to Sheets

4. `@RestController` vs `@Controller`

Spring REST can handle all standard **HTTP methods** (Verbs) as defined by the REST architectural style:

- **GET** (Retrieve data)
- **POST** (Create new resource/submit data)
- **PUT** (Update/Replace an entire resource)
- **PATCH** (Update/Modify part of a resource)
- **DELETE** (Remove a resource)
- **HEAD** (Retrieve headers only)
- **OPTIONS** (Retrieve allowed communication options)

5. How Each HTTP Method Can be Handled

Each HTTP method is typically handled using its specific, corresponding annotation:

HTTP Method	Spring Annotation	Typical Use Case
`GET`	`@GetMapping`	`/users?name=John&active=true`
POST	`@PostMapping`	`/users` (create a new user)
PUT	`@PutMapping`	`/users/{id}` (update the entire user)
DELETE	`@DeleteMapping`	`/users/{id}` (delete a user)
PATCH	`@PatchMapping`	`/users/{id}` (update a single field, e.g., email)

Export to Sheets

6. What is `ResponseEntity`?

`ResponseEntity` is a class in Spring that represents the entire HTTP response, including the **body, status code,** and **headers**.

- **How to Write It:** You wrap your data within a `ResponseEntity` object:

```
@GetMapping("/{id}")
public ResponseEntity<Object> getUser(@PathVariable Long id) {
    User user = findUserById(id);
    if (user == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<>(user, HttpStatus.OK);
}
```

- **Information Conveyed:** `ResponseEntity` allows you to convey all crucial aspects of the HTTP response:
  - **Data (Body):** The generic type, e.g., `<User>`.
  - **HTTP Status Code:** Essential for conveying the result of the operation (e.g., `200 OK`, `201 CREATED`, `404 NOT FOUND`).
  - **Headers:** Additional information like `Location` (for newly created resources), `Cache-Control`, or custom headers.

7. What is HTTP Status Code?

An HTTP Status Code is a three-digit integer code returned by a server to a client in response to an HTTP request. The code's primary purpose is to convey the **result** and **status** of the request (whether it was successful, redirected, or failed).

Status Code	Class	Meaning
200 OK	Success	The request has succeeded. Standard successful response.
201 Created	Success	The request has succeeded and a new resource has been **created**.
400 Bad Request	Client Error	The server cannot process the request due to an error on the client side (e.g., validation failure, JSON error).
401 Unauthorized	Client Error	The request requires user authentication (user is not logged in).
403 Forbidden	Client Error	The server understand the request but refuses to authorize it (user is logged in but lacks permissions).
404 Not Found	Client Error	The server cannot find the requested resource.
500 Internal Server Error	Server Error	A generic error side error occurred while processing the request.

Export to Sheets

8. Global Exception Handling

Global Exception Handling is the centralized mechanism in Spring to manage exceptions thrown across all controllers in an application.

- `@ControllerAdvice`: This annotation marks a class that will provide common error-handling logic for controllers throughout the application. It is essentially a controller advice (interceptor) for controllers.
- `@ExceptionHandler`: This annotation is placed on methods within the `@ControllerAdvice` class. It specifies which exception type the method is responsible for handling.
- **Benefit:**
  - **Decoupling:** Separates error handling logic from business logic in the controllers.
  - **Consistency:** Ensures a uniform response structure (e.g., JSON error object) and standard HTTP status code for all errors of a specific type.
  - `RFC Client Request Mismatch` Needs setting `ignoreUnknownProperties` to `false`.

9. When to Use Spring MVC and Spring REST

- **Use Spring MVC:** When you are building a **monolithic, server-side rendered web application** where the server generates and returns HTML pages directly to the browser (e.g., using Thymeleaf, JSP). This is typically used for traditional websites or internal tools.
- **Use Spring REST:** When you are building a **backend API** that will be consumed by separate, decoupled clients (e.g., a Reactive/React single-page application, mobile app, or another microservice). This is the standard approach for modern, distributed architectures.

10. Practical Example for Spring REST

Scenario: A simple Customer Management API

Endpoint	HTTP Method	Action
`/api/customers`	**POST**	Create a new customer record.
`/api/customers/{id}`	**GET**	Retrieve customer details by ID.
`/api/customers/{id}`	**PUT**	Replace a list of all customers.

Export to Sheets

Example Controller Code:

```
// app
// ...
@RestController
@RequestMapping("/api/customers")
public class CustomerController {

    @Autowired
    private CustomerService customerService;

    // POST: Create a new customer (Returns 201 CREATED)
    @PostMapping
    public ResponseEntity<CustomerDTO> createCustomer(@RequestBody CustomerDTO cust
        CustomerDTO savedDto = customerService.createCustomer(dto);
        return new ResponseEntity<>(savedDto, HttpStatus.CREATED);
    }

    // GET: Retrieve a customer (Returns 200 OK or 404 NOT FOUND)
    @GetMapping("/{id}")
    public ResponseEntity<CustomerDTO> getCustomer(@PathVariable Long id) {
        CustomerDTO dto = customerService.getCustomerById(id);
        return ResponseEntity.ok(customer); // Returns 200 OK and the body
    }
}
```

12. Layered Architecture

A **Layered Architecture** organizes an application's code into distinct, horizontal layers, each with specific responsibilities. Spring REST applications commonly use 3 layers:

1. **Controller Layer (API/Web Layer):**
   - **Responsibility:** Handling incoming HTTP requests, mapping URLs, parsing request parameters, invoking the Service Layer, and formatting the outgoing HTTP response (e.g., JSON/Value conversion).
   - **Key Annotations:** `@RestController`, `@RequestMapping`, `@GetMapping`, etc.
2. Service Layer (Business Logic Layer):
   - **Responsibility:** Containing the core business rules and transaction management, coordinate multiple DAO/Repository calls and performs domain-specific calculations.
   - **Key Annotations:** `@Service`, `@Transactional`
3. DAO/Repository Layer (Data Access Layer):
   - **Responsibility:** Interacting directly with the database. It contains logic for CRUD (Create, Read, Update, Delete) operations and maps data to/from the database entities.
   - **Key Annotations:** `@Repository` (often extends Spring Data JPA's `JpaRepository`).

13. How to Test Individual Layers

Testing each layer in isolation is crucial for quality:

1. **Controller Layer Testing (Unit/Slice):**
   - Use `@WebMvcTest` to slice test annotated with `@Mvc` to test how the controller handles requests, inputs, and outputs. You typically **mock** the Service Layer dependency to isolate the controller's logic.
2. **Service Layer Testing (Unit):**
   - Use JUnit and Mockito to test the business logic within the service class. You mock the Repository Layer dependency to isolate the service's logic.
3. **Repository Layer Testing (Integration/Slice):**
   - Use `@DataJpaTest` to slice test annotation to test the persistence logic. This uses an in-memory database (like H2) to verify that the repository correctly interacts with the database (saves, queries, updates entities).

14. When to Use `@RestController` and `@Controller`

- Use `@RestController` when the class is designed to handle **RESTful APIs** and every method should return data (e.g., JSON/XML) directly to the client's body. This is the **default for Spring REST**.
- Use `@Controller` when the class is part of a traditional Spring MVC application and is intended to return a logical **View Name** (e.g., a string like `"home"`) that gets resolved to an HTML template or a `ModelAndView` object. If a `@Controller` method needs to return data directly, you must additionally annotate it with `@ResponseBody`.

15. What is DTO?

A DTO (Data Transfer Object) is a simple data object used to transfer data between layers of an application (e.g., between the Controller and the Service layer, or between the Service and the client).

- **Difference from Entity Class:**
  - **Entity Class:** Represents a table/record in the database. It often contains JPA annotations (`@Entity`, `@Id`) and is coupled to the persistence mechanism. It represents the full data model.
  - **DTO Class:** Represents a specific **subset** of data needed for a specific operation. It is **decoupled** from the database and only contains fields necessary for the client or layer interaction.
- **Conversion:** The conversion from **Entity** → **DTO** (for sending data to the client) and **DTO** → **Entity** (for saving data to the database) is handled in the Service Layer using manual mapping code, or more commonly, with a dedicated mapping library like **MapStruct** or **ModelMapper**.

16. Difference Between `@Response Mapping` and `@Get Mapping`

- `@RequestMapping`: The original and more generic annotation. It can be applied at the class or method level and requires you to **write the HTTP method** (e.g., `@RequestMapping(method = RequestMethod.GET)`). It can handle any HTTP method.
- `@GetMapping`: A **convenience/shortcut** annotation introduced in Spring 4.3. It is simply a composed annotation equivalent to `@RequestMapping(method = RequestMethod.GET)`. It is **cleaner and highly recommended** for readability.

The same applies to `@PostMapping`, `@PutMapping`, etc.

17. How Versioning of the Product is Handled

Versioning is critical for existing REST APIs without breaking existing clients. Spring supports three main versioning strategies:

1. **URI (Path) Versioning (Most Common):** Embeds the version number into the URL path.
   - **Example:** `@GetMapping("/api/v1/users")` and `@GetMapping("/api/v2/users")`
2. **Request Parameter Versioning:** Uses a query parameter to specify the desired version.
   - **Example:** `@GetMapping("/api/users", "version=1")`
3. Header Versioning (Accept Header): Uses a custom request header or the standard `Accept` header to specify the version (Media Type versioning).
   - **Example:** `@GetMapping("/api/users", headers = "X-API-VERSION=1")`

18. What is HATEOAS?

HATEOAS (Hypermedia As The Engine Of Application State) is a component of the REST architectural style.

- It means that the API response should **include links** that inform the client what actions can be taken next, essentially driving the application state through hypermedia.
- **Benefit:** Makes the API self-discoverable. A client only needs to know the initial entry point, and the API guides it through the possible operations (e.g., a user resource response includes a link to update the user or view their orders).
- In Spring: Implemented using the **Spring HATEOAS** library.

19. What is Swagger?

Swagger (now part of the larger **OpenAPI Specification**) is a set of tools and standards for defining, describing, consuming, and visualizing RESTful web services.

- **Why we need Swagger:** It serves as **documentation** for the API.
  - **Interactive Documentation:** It generates a beautiful, browser-based, interactive UI (Swagger UI) from the application's code and annotations, allowing developers to read and test the API endpoints directly.
  - **Contract Definition:** It provides a language-agnostic interface description for the API, allowing client teams to start development even before the backend is fully complete.
- In Spring: Typically integrated using **Springdoc-openapi** or **Springfox**.

20. Purpose of Dev Tools in Spring Boot

Spring Boot DevTools (`spring-boot-devtools`) is a module that provides several development-friendly tools to improve the development experience and productivity:

- **Automatic Restart:** The main feature. DevTools automatically restarts the application whenever files on the classpath change. It uses two classloaders to achieve "live reload" functionality, making the restart much faster.
- **LiveReload:** Automatically triggers a browser refresh when resources (like HTML, CSS) are updated.
- **Default Properties:** Provides sensible development time configuration defaults (e.g., template caching is disabled for immediate visual feedback).
- **Remote Debugging:** Supports remote tunneling for debugging deployed applications (through used cautiously).

Export to Sheets

Spring Boot
1. What is Spring Boot. How is it different from traditional spring framework.

Here are the answers to your questions about Spring Boot:

1. What is Spring Boot?

Spring Boot is an opinionated, convention over configuration framework built on top of the Spring ecosystem.

Spring applications by eliminating much of the cumbersome manual configuration. It enables developers to create **stand-alone, production-grade Spring applications** that can be "just run". 🚀

Difference from Traditional Spring Framework

Feature	Spring Boot	Traditional Spring Framework
Configuration	**Minimized** or **Zero** configuration, relies heavily on conventions and auto-configuration.	**Extensive** configuration required, typically using XML files or verbose Java Config.
Dependencies	Uses **Starter Dependencies** to group common dependencies together (e.g., `spring-boot-starter-web`).	Requires manually defining versions and managing compatibility for many libraries.
Deployment	Includes an **Embedded Server** (Tomcat, Jetty, Undertow) by default, allowing for easy stand-alone JAR file.	Requires an external application server (like Tomcat or JBoss) to be installed and configured as a WAR file.
Setup Time	Very **fast**; allows developers to start writing business logic almost immediately.	**Slow** and complex setup process.

Export to Sheets

2. What are the Benefits of Spring Boot?

The key benefits of using Spring Boot include:

1. **Auto-Configuration:** Automatically configures your application based on the JARs present in the classpath, reducing boilerplate configuration code (e.g., detects JPA and configures a data source automatically).
2. **Starter Dependencies:** Simplifies build configuration by providing a set of curated and tested dependencies in one place.
3. **Embedded Servers:** Simplifies deployment by embedding a container (Tomcat, Jetty) right into the executable JAR.
4. **Production-Ready Features:** Includes non-functional features like Metrics, Health Checks, and Externalized Configuration (via **Spring Boot Actuator**).
5. No XML Configuration: Favors convention over configuration, allowing you to write purely Java-based Spring applications.

3. What is Actuator and its URLs?

Spring Boot Actuator is a sub-project that provides production-ready features to **monitor and manage** your application when it's pushed to a production environment.

Actuator URL (Endpoint)	Purpose	Benefit
`/actuator/health`	Displays basic application health information (e.g., up/down status).	Essential for **load balancers** and **Kubernetes** (liveness/readiness probes) to check if an instance is alive and should receive traffic.
`/actuator/info`	Displays arbitrary application information (e.g., custom build version, Git commit).	Allows operators to quickly determine **which version** of the application is running.
`/actuator/metrics`	Shows available metrics for the current application, including JVM memory, CPU usage, HTTP request counts, and garbage collection details.	Crucial for **monitoring tools** like Prometheus/Grafana to track application performance, resource bottlenecks, and set up alerts.
`/actuator/beans`	Displays a complete list of all the Spring beans loaded in the application context.	Useful for **debugging** dependency injection issues or verifying that all components are correctly configured and loaded.
`/actuator/env`	Displays all environment variables and configuration properties that Spring has loaded.	Helps **diagnose configuration problems** by showing exactly which environment variable or file is being used by the command line, system variables.

Export to Sheets

4. 20 Starters for Spring Boot Application

A **Spring Boot Starter** is a convenience dependency bundle that includes all the common transitive dependencies needed to get a feature up and running quickly.

Category	Starter Dependency Name	Purpose
Web	`spring-boot-starter-web`	Develop web applications and RESTful APIs (includes Tomcat and Spring MVC).

Web	`spring-boot-starter-web`	Develop machine web applications using Spring MvcFlux
	`spring-boot-starter-jersey`	Default embedded Tomcat container
Data	`spring-boot-starter-data-jpa`	Use Spring Data JPA with Hibernate for relational databases.
	`spring-boot-starter-jdbc`	Use JDBC with a connection pool.
	`spring-boot-starter-data-mongodb`	Connect to and interact with MongoDB.
	`spring-boot-starter-data-redis`	Integrate with Redis key-value stores.
Testing	`spring-boot-starter-test`	Includes JUnit, Mockito, Spring Test, and Assertj for unit and integration tests.
Production	`spring-boot-starter-actuator`	Provides production-ready monitoring and management endpoints.
Security	`spring-boot-starter-security`	Configure Spring Security for authentication and authorization.
Messaging	`spring-boot-starter-amqp`	Integrate with AMQP (Advanced Message Queuing Protocol).
	`spring-boot-starter-kafka`	Integrate with Apache Kafka messaging system.
Templating	`spring-boot-starter-thymeleaf`	Use the Thymeleaf templating engine for MVC views.
Caching	`spring-boot-starter-cache`	Provides support for Spring's caching abstraction.
Configuration	`spring-boot-starter-validation`	Includes Hibernate Validator for bean validation.
	`spring-boot-starter-logging`	Default logging starter (using Logback).
Client	`spring-boot-starter-oauth2-client`	OAuth 2.0/OpenID Connect client support.
Cloud	`spring-boot-starter-config-client`	Connect to a Spring Cloud Config Server.
Utilities	`spring-boot-devtools`	Development tools for faster iteration (automatic reload, LiveReload).
GraphQL	`spring-boot-starter-graphql`	Build GraphQL services.

Export to Sheets

5. What is `@SpringBootConfiguration`?

The `@SpringBootConfiguration` annotation is a **convenience annotation** that marks the main class of a Spring Boot application. It is a composable annotation that combines three necessary annotations:

1. `@Configuration`
2. `@EnableAutoConfiguration`
3. `@ComponentScan`

Annotation	Meaning	Purpose
`@SpringBootConfiguration`	`@Configuration`	Marks the class as a source of bean definitions for the Spring IoC container.
`@EnableAutoConfiguration`	The core of Spring Boot.	**Enables auto-configuration**. Tells Spring to automatically configure beans based on the JAR dependencies present in the classpath.
`@ComponentScan`	Scans for components.	**Enables component scanning**. Tells Spring to look for and register components (like `@Service`, `@Repository`, `@Controller`, etc.) starting from the package where this class is located (the base package).

Export to Sheets

6. Free Properties of `application.properties`

The `application.properties` (or `application.yml`) file is the default place for Spring Boot applications to externalize configuration.

Property Key	Description	Default Value
`server.port`	The port the embedded web server should listen on.	`8080`
`spring.application.name`	The name of the application, used for monitoring and service discovery.	`application`
`logging.level.root`	The root level for logging (e.g., INFO, DEBUG, WARN, ERROR).	`INFO`
`spring.datasource.url`	The JDBC URL for connecting to the database.	(None)
`spring.datasource.username`	The database username.	(None)
`spring.jpa.hibernate.ddl-auto`	Controls how Hibernate manages the database schema (e.g., `none`, `update`, `create`, `validate`).	`none` (default for production) `update` (common for development)
`spring.jpa.show-sql`	Whether to log SQL statements to the console.	`false`
`management.endpoints.web.exposure.include`	A comma-separated list of Actuator endpoints to expose over HTTP (e.g., `health,info,metrics`).	`health,info`

Export to Sheets

What is repository
2. What are the different repositories in the Spring data jpa...

Here are the answers to your questions about Spring Data JPA, Hibernate, and related concepts.

1. What is Repository?

A Spring `Repository` (often known as the Data Access Object or DAO) is a pattern and a component responsible for abstracting the storage of data persistence. It acts as a collection-like interface for managing domain objects (entities) and communicating with the DBMS (Create, Read, Update, Delete) operations and complex queries against the underlying database.

2. Different Repositories in Spring Data JPA

Spring Data JPA provides several repository interfaces that extend each other, offering increasing levels of functionality:

1. `Repository`: The base marker interface.
2. `CrudRepository`: Provides basic CRUD functionalities.
3. `PagingAndSortingRepository`: Extends `CrudRepository` to add methods for pagination and sorting.
4. `JpaRepository`: Extends `PagingAndSortingRepository` to add JPA specific features like flushing and methods for batch deletion.

`JpaRepositoryImplementation`: Not typically part of the main hierarchy, but can be added to repositories to execute queries based on the JPA Criteria API.

3. Functions of Key Repository Interfaces

Repository Interface	Key Functions
`CrudRepository`	`save(T entity)`, `findById(ID id)`, `findAll()`, `delete(T entity)`, `deleteAll()`, `count()`, `existsById(ID id)`
`PagingAndSortingRepository`	`findAll(Sort sort)`, `findAll(Pageable pageable)`: Retrieves entities in pages (for pagination).
`JpaRepository`	`flush()`: Synchronizes persistence context with the database. `saveAndFlush(T entity)`: Combines `save` and `flush`. `deleteInBatch(Iterable<T> entities)`, `deleteAllInBatch()`: Optimized batch deletion.

Export to Sheets

4. What is Dialect?

A **Dialect** is a component of Hibernate that is configured in your `application.properties` file. It translates **types and SQL statements to the specific syntax and features of a particular database** vendor.

Since databases like MySQL, PostgreSQL, and Oracle have slightly different SQL syntax, data types, and function names, the dialect tells Hibernate which specific SQL to generate (e.g., how to generate a primary key, or what function to use for current date).

5. Database Connection Properties in `application.properties`

To connect a Spring Boot application to a relational database, you typically need to specify these properties:

```
# 1. Database Connection URL
spring.datasource.url=jdbc:mysql://localhost:3306/myDatabaseName?useSsl=false

# 2. Database Credentials
spring.datasource.username=root
spring.datasource.password=secret

# 3. Driver Class Name (Optional for modern Spring Boot, but good practice)
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# 4. JPA/Hibernate Configuration (The Dialect and DDL)
spring.jpa.hibernate.ddl-auto=update
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
```

6. What is Named Query?

A **Named Query** is a JPA/Hibernate query defined once, typically in an **annotation** on the entity class or in an **XML file**, and then executed later by referring to its name.

- **Benefit:** It externalizes the query from the Java code, improving code clarity and allowing the JPA provider to **parse and optimize the query once** when the application starts.

7. What is Native Query? How to Write It?

A **native query** is a query written in the **actual SQL syntax** of the target database (e.g., MySQL's specific SQL). It is used when JPQL or the Criteria API cannot express the required complex query, or when optimizing for database-specific features.

How to Write It (using `@Query` in a Repository):

```
Java

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    @Query(value = "SELECT * FROM emp_table WHERE age > ?1", nativeQuery = true)
    List<Employee> findOlderThan(int age);
}
```

The key is setting the `nativeQuery` attribute to `true`.

8. What is JPQL? How to Write It?

JPQL (Java Persistence Query Language) is a platform-independent, object-oriented **query language** defined as part of the JPA specification.

- **Features:** JPQL queries work with **Entities and their fields** (Java objects) rather than database tables and columns. It is translated by Hibernate into the target database's specific SQL.

How to Write It (using `@Query` in a Repository):

```
Java

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    // JPQL using positional parameters
    @Query("SELECT e FROM Emp e WHERE e.salt > ?1")
    List findHighSalaries(Integer salt);

    // JPQL using named parameters (preferred)
    @Query("SELECT e FROM Emp e WHERE e.name = :name")
    Emp findByUsernameQueryParam(@Param("name") String username);
}
```

9. Differences and Usage of Query Types

Query Type	Based On	Portability	When to Use
Spring Data Method Names	Entity fields	High	For simple CRUD and common lookups (e.g., `findByLastName`).
`@Query` (JPQL)	Entity fields and relationships	High (database agnostic)	For custom queries where performance or complex logic is required.
Native SQL (`@Query(nativeQuery=true)`)	Database-specific SQL	Low (database-specific)	For complex, highly optimized queries or when using database-specific features.

Export to Sheets

10. Difference Between Hibernate and Spring Data JPA

- **JPA (Java Persistence API): A specification/standard** for managing Java objects in a relational database. It defines a set of interfaces, annotations, and rules.
- **Hibernate:** A complete, mature **implementation** of the Java Persistence API specification. It contains additional features and logic to perform the ORM (Object-Relational Mapping).
- **Spring Data JPA: A convenience abstraction** layer built on top of the JPA specification (and thus, on providers like Hibernate). It drastically reduces boilerplate code by automatically implementing the Repository interfaces.

Summary: Spring Data JPA doesn't replace Hibernate. It makes using Hibernate (or the JPA provider) much simpler.

11. Benefits of Using Spring Data JPA over Hibernate

The primary benefit is **developer productivity** and **reduced boilerplate code**:

1. **Automatic Repository Implementation:** Spring Data JPA writes the full implementation for CRUD and common query methods based purely on the repository interface definition.
2. **Query Derivation:** It automatically generates queries based on method names (e.g., `findByUsernameAndEmail`).
3. **Clean Abstraction:** It provides a high-level, consistent programming model regardless of the underlying JPA provider.
4. **Pagination and Sorting:** Provides easy, built-in methods for handling complex pagination and sorting requirements.

12. Architecture of Spring Data JPA

The architecture is layered:

1. **Repository Interface:** Defined by the developer (e.g., `UserRepository`).
2. **Proxy Bean:** Creates a **proxy implementation** of the repository interface at runtime. It translates method calls (`findBy...`) into JPA queries.
3. **JPA API:** The standard interface used by Spring Data JPA.
4. **JPA Provider (e.g., Hibernate):** The actual implementation that processes the JPA calls.
5. **JDBC Driver:** Used by the JPA provider to communicate with the database.

`Database` The persistence storage.

Entity Manager: Spring Data JPA internally uses the `EntityManager` to perform all database operations, delegating its management to the Spring Framework.

13. What is Entity Manager in Spring Data JPA?

The `EntityManager` is the **core JPA component** responsible for interacting with the persistence context (the environment where entities are managed). It is the **interface** used to perform all persistence operations.

- **Functions:** It handles creating, reading, updating, deleting (CRUD) entities, managing transactions, executing queries (JPQL, native), and handling entity state transitions (e.g., detached → managed).
- **In Spring Data JPA:** Developers rarely interact with the `EntityManager` directly. Spring Data JPA handles its lifecycle and automatically injects it into the repository implementation.

14. Entity Annotations

Annotation	Level	Purpose
`@Entity`	JPA	Marks a Java class as a persistent entity, representing a table in the database.
`@Table`	JPA	Specifies the name of the database table to which this entity is mapped (optional if class name matches table name).
`@Id`	JPA	Designates the field that serves as the **primary key** of the entity.
`@GeneratedValue`	JPA	Specifies how the primary key value is generated (e.g., `IDENTITY`, `SEQUENCE`).
`@Column`	JPA	Specifies the details of the database column mapped by the field (e.g., name, length, nullable).
`@Transient`	JPA	Marks a field that should **not** be persisted to the database. It exists only in the Java object.
`@Query`	Spring Data	Defines a custom query (JPQL or Native SQL) to be executed by the repository method.
`@NamedQuery`	JPA	Defines a query string once on the entity class, allowing it to be executed later by name.

Export to Sheets

15. Difference Between JPQL and Native Query

Over-DP for a quick table: this provides further context.

Feature	JPQL	Native Query (Native SQL)
Syntax	Object-Oriented (refers to entities and fields)	Database-Specific SQL (refers to tables and columns)
Portability	**High:** Works across different database vendors.	Not high. Tied to a specific database (e.g., using Oracle, SQL).
Safety	Safe. Checked by the JPA parser for entity existence and syntax correctness.	Less safe. Errors show up at runtime unless manually validated.
When to Use	Use for **most custom queries** where the structure of the data is defined by the entities.	Use for **optimization**, calling database-specific functions, or complex joins that are difficult in JPQL.

Export to Sheets

16. Mappings in Spring Data JPA

Spring Data JPA uses standard JPA annotations for defining object-relational mappings (relationships):

Mapping Type	Relationship	Annotation	Code Example (in the Parent Entity)
One-to-One	1:1	`@OneToOne`	`public class User { @OneToOne private UserProfile profile; }` (Typically, you'd provide address, address;)
One-to-Many	1:N	`@OneToMany`	`public class User { @OneToMany(mappedBy = "user") private List<Post> posts; }`
Many-to-Many	M:N	`@ManyToMany`	`public class User { @ManyToMany @JoinTable(name = "user_role") private Set<Role> roles; }`

Export to Sheets

Key Points:

- `mappedBy`: Used on the **non-owning** (inverse) side of the relationship (typically the "Many" side in 1:N or one side in M:N) to specify the field in the owning entity that manages the relationship.
- `@JoinColumn`: Used on the **owning** side (where the foreign key column resides).
- `@JoinTable`: Used in M:N to specify the name of the intermediate join table.

17. What is `@Transactional`?

The `@Transactional` annotation is a declarative way to manage transactions in Spring. When applied to a class or method, Spring automatically manages the transaction lifecycle (starting, committing, and rolling back).

Benefit:

If you use this annotation, you get the following benefits related to transactions:

1. **Atomicity Guarantee:** Ensures that all database operations within the method are treated as a single unit (either all succeed and commit, or if any fail, all are rolled back).
2. **Exception Handling:** By default, the transaction will automatically **roll back** on unchecked exceptions (`RuntimeException`) and automatically **commit** on successful execution or checked exceptions.
3. **Simplified Code:** You avoid writing manual transaction boilerplate (`beginTransaction()`, `commit()`, `rollback()`) across your Service Layer.
4. **Isolation:** Allows configuring the transaction's isolation level (e.g., `READ_COMMITTED`) to control concurrent access between
5. **Propagation:** Allows configuring the transaction's behavior when it is called from another transaction (e.g., `REQUIRED`, `REQUIRES_NEW`).

18. What are ACID Properties

ACID is an acronym defining the set of properties that guarantee that database transactions are processed reliably.

Property	Full Name	Explanation
A	Atomicity	The entire transaction is treated as a single, indivisible unit. Either all of its changes are performed (committed), or none of them are (rolled back).
C	Consistency	A transaction must ensure that the database moves from one **valid state** to another. It maintains all defined rules, constraints (primary keys, foreign keys), and triggers.
I	Isolation	Ensures that concurrent transactions execute in isolation, as if they were running sequentially. The intermediate state of a transaction is **invisible** to other transactions.
D	Durability	Once a transaction has been committed, it will remain permanently recorded in the database, even in the event of a system failure (e.g., power loss).

Export to Sheets

Spring Microservices...
1. What is a Microservice.
2. What is monolithic architecture...

Here are the answers to your questions about Spring Microservices, JUnit, and Mockito.

Spring Microservices

1. What is a Microservice?

A **Microservice** is an architectural approach where a large application is broken down into a collection of smaller, independent services. Each service:

- Runs in its own process.
- Is developed and deployed **independently**.
- Focuses on a **single business capability** (e.g., Inventory, User Management).

message queue.

3. What is Monolithic Architecture?

Monolithic Architecture is a traditional application structure where the entire application—including all business logic, data access, and user interface logic—is built as a **single, tightly coupled unit**.

- All components share the same codebase and are deployed together as one large package (e.g., a single WAR or EAR file) on a single server or cluster.

5. How Microservice Architecture is Different from Monolithic Architecture?

Feature	Microservice Architecture	Monolithic Architecture
Deployment	Independent and frequent deployment of small services	Single, integrated deployment of the entire application
Scaling	**Independent scaling** via high demand services/resource-intensive	**Entire application** must be scaled, even if only one component needs it
Technology	Polyglot persistence/programming: teams can use the best technology for the job.	Single technology stack for the entire application
Fault Tolerance	Better. failure in one service only affects that service.	Lower. failure in one part can bring down the entire application.
Development	Smaller, independent teams work on their own services.	Large teams write a large, complex codebase.

Export to Sheets

6. What is Eureka Server? Why do we need it?

The **Eureka Server** is Spring Cloud Netflix's implementation of a **Service Discovery** server.

- **Service Discovery** in a microservice environment, services are constantly scaling up and down, and their network locations (IP addresses and ports) change frequently. A service needs a reliable way to find the network location of another service.
- **Eureka's Role**: The Eureka server acts as a **central registry**. Client microservices (Eureka Clients) register themselves with the Eureka server upon startup, and other services query the server to find the location of a dependency service.

7. How One Microservice Can Communicate with Another Microservice?

Microservices communicate primarily through the network using standard protocols. The two most common methods are:

1. **Synchronous Communication (REST/HTTP)**
   - One service (Client) makes a direct, blocking HTTP request to another service (Server) and waits for a response.
   - **Implementation**: Typically uses Spring's `RestTemplate`, `WebClient` (**Reactive**), or the declarative **Feign Client** (recommended).
2. **Asynchronous Communication (Messaging)**:
   - Services communicate via a message broker (e.g., RabbitMQ, Kafka). A service sends an event/message and doesn't wait for an immediate response.
   - **Implementation**: Typically uses **Spring Cloud Stream** with a binder for the broker.

8. What is Feign Client?

The **Feign Client** (or OpenFeign) is a **declarative HTTP client** developed by Netflix and integrated into Spring Cloud.

- Purpose: It dramatically simplifies writing HTTP clients for REST communication between services.
  serialization, and message exceptions, you simply define an **interface** with method annotations.
- **Mechanism**: Feign dynamically generates the implementation of that interface at runtime, handling the actual HTTP communication transparently.

7. Dependencies Required for Feign Clients and Eureka Clients

These are typically added as Spring Boot Starter dependencies in your project's `pom.xml` (Maven) or `build.gradle` (Gradle):

Component	Spring Cloud Starter Dependency	Key Annotation in Application
Eureka Client	`spring-cloud-starter-netflix-eureka-client`	`@EnableEurekaClient` or `@EnableDiscoveryClient` (In the service that needs to register/discover)
Feign Client	`spring-cloud-starter-openfeign`	`@EnableFeignClients` (In the service that needs to use Feign)

Export to Sheets

JUnit and Mockito

1. What is JUnit?

JUnit is the most popular, open-source testing framework for Java.

- **Purpose**: It provides annotations (like `@Test`, `@BeforeEach`) and assertions (like `assertEquals`) to help developers write and run **Unit Tests**. Unit tests verify that individual components (units) of source code—usually a single method or class—are working correctly in isolation.

2. What is the Purpose of Mockito?

Mockito is a popular Java mocking framework.

- **Purpose**: Its main goal is to create **mock objects** for dependencies during unit testing.
- **Why Mock**: In a true unit test, you only want to test the logic of the class under test (the Service). You don't want to hit a database or a third-party API. Mocking allows you to simulate the Repository, a third-party API, using a real dependency would turn the unit test into an integration test. Mockito creates mock objects that:
  1. **Simulate** the behavior of real dependencies (e.g., tell the mock repository to return a specific user when a specific ID is passed).
  2. **Verify** that the SUT interacts with its dependencies correctly (e.g., verify that the service called the repository's `save()` method exactly once).

> 1. What is Angular and why is it used?
> 2. How do you create a new Angular project using the CLI?

Here are the answers to your questions about Angular.

1. What is Angular and Why is it Used?

Angular is a powerful, open-source **TypeScript-based framework** for building **client-side, single-page applications (SPAs)**. It is maintained by Google. ✓ ✓

It is used because it provides a complete, structured platform for building large, scalable applications, offering features like:

- **Component-based Architecture**: Facilitates modularity and reusability. ✓
- **Two-Way Data Binding**: Simplifies synchronization between the model and the view. ✓
- **Comprehensive Tooling**: The Angular CLI offers excellent generation and build tools. ✓
- **TypeScript**: Provides static typing, which improves code quality and maintainability. ✓

2. How Do You Create a New Angular Project using the CLI?

You create a new Angular project using the Angular CLI command `ng new`, followed by the desired project name: ✓

Bash

```
ng new project-name
```

The CLI will prompt you for configuration details like routing setup and stylesheet format (CSS, SCSS, etc.), and then install all necessary dependencies.

3. What is a Component in Angular?

A **Component** is the most fundamental building block of an Angular application. It controls a part of the screen or a **view**. ✓

Every component consists of three parts: ✓

- **A TypeScript Class**: To handle data and logic. ✓
- **An HTML Template**: To define the view/markup. ✓
- **A CSS/Style File**: To define the component's appearance. ✓
- **A Metadata** defined by the `@Component` decorator: Ex link the class to its template and styles. ✓

4. What is the Purpose of `@Component` Decorator?

The `@Component` **decorator** is a function that attaches **metadata** to a TypeScript class, officially declaring it as an Angular component. ✓

The metadata is crucial as it tells Angular how to process the class, template, and styles. Key properties include: ✓

- `selector`: The CSS selector used to instantiate the component in a template (e.g., `'app-header'`). ✓
- `templateUrl` or `template`: The path to the component's HTML file or the inline HTML markup. ✓
- `styleUrls` or `styles`: An array of paths to the component's CSS files or inline styles. ✓

5. How Does Data Binding Work in Angular?

Data binding is the mechanism that synchronizes the data between the **Component's** TypeScript class (Model) and the Template (View). ✓

Angular supports four main forms of binding:

1. **Interpolation** (`{{ data }}`): One-way from **Model to View**.
2. **Property Binding** (`[property]="data"`): One-way from **Model to View** (sets an HTML element property).
3. **Event Binding** (`(event)="handler()"`): One-way from **View to Model** (responds to user actions).
4. **Two-Way Binding** (`[(ngModel)]="data"`): Combines property and event binding for automatic synchronization. ✓

6. What is Interpolation in Angular Templates?

Interpolation is a one-way data binding technique used to display a component's property value as a string in the HTML template. ✓

It uses **double curly braces** as delimiters:

HTML

```
<h1>Hello, {{ userName }}!</h1>
```

7. What is Property Binding in Angular?

Property Binding is a one-way data binding technique used to set an HTML element's property to the value of a component's property. The target HTML element property is enclosed in **square brackets** `[]`. ✓

HTML

```
<img [src]="imageUrl" [alt]="imageAlt">
```

8. What is Event Binding in Angular?

Event Binding is a one-way data binding technique that allows the component to listen for and respond to events from the DOM elements in the view (e.g., clicks, key presses). The event name is enclosed in **parentheses** `()`. ✓

HTML

```
<button [disabled]="buttonDisabled"(click)="myFunction">
```

9. What is Two-Way Data Binding?

Two-Way Data Binding is a mechanism that automatically synchronizes data flow between the component's model (TypeScript class) and the view (HTML input elements). ✓

It is achieved using the `[(ngModel)]` directive, often referred to as the "**banana in a box**" syntax. ✓

- It combines **Property Binding** (setting the input value) and **Event Binding** (listening for input changes) into a single syntax.
- **Note**: Using `ngModel` requires importing the `FormsModule` into the application module.

HTML

```
<input [(ngModel)]="searchTerm">
```

10. What is the Role of `NgModule`?

The `NgModule` (Angular Module) defines a **logical boundary** or grouping for related application components, services, and other code. It's the central mechanism for structuring and organizing an Angular application.

- The `NgModule` decorator is used to attach metadata to a class, specifying: ✓
  - `declarations`: Components, Pipes, and Directives that belong to this module.
  - `imports`: Other `NgModule`s whose exported classes are needed by templates in this module. ✓
  - `providers`: Services that this module contributes to the global collection of services (the dependency injection system).
  - `bootstrap`: The root component that Angular should load when it starts (only in the root module).

11. What is the Difference Between `ngIf` and `ngFor`?

Both `ngIf` and `ngFor` are **Structural Directives** (meaning they change the DOM structure), but they serve different purposes: ✓

Directive	Purpose	Mechanism	Usage
`ngIf`	**Conditional Rendering**: Adds or removes an element (and its children) from the DOM based on a boolean expression.	If false, the element **is physically** removed from the DOM.	`<div *ngIf="isLoggedIn">...</div>`
`ngFor`	**List Rendering**: Repeats a portion of the DOM template for each item in an iterable/collection (like an array).	Creates a distinct instance of the template for each item in the list.	`<li *ngFor="let item of items">...</li>`

Export to Sheets

12. What is a Directive in Angular?

A **Directive** is a class that adds behavior to an existing DOM element or component. It allows developers to extend HTML's capabilities.

Angular has three main types of directives: ✓

1. **Component Directives**: Directives with a template, which are the main building blocks of the UI (covered in Q3). ✓
2. **Structural Directives**: Change the DOM layout by adding, removing, or manipulating elements (e.g., `*ngIf`, `*ngFor`, `*ngSwitch`). ✓
3. **Attribute Directives**: Change the appearance or behavior of an existing element, component, or another directive (e.g., `ngModel`, `ngStyle`). ✓

13. What is a Pipe in Angular?

A **Pipe** is a class used to transform data before it is displayed in the view. They are used directly within the template using `|`.

- **Syntax**: `{{ value | pipeName[:arg1[:arg2]] }}`
- **Examples**: Built-in pipes include `DatePipe`, `CurrencyPipe`, `UpperCasePipe`, and `JsonPipe`. You can also create custom pipes. ✓

HTML

```
<p>Today's date is: {{ today | date:'shortDate' }}</p>
```

14. How Do You Use the Async Pipe?

The `async` **pipe** is an **impure pipe** used to automatically subscribe to an **Observable** or **Promise** and return the latest value it has emitted.

- **Purpose**: It handles the subscription and unsubscription automatically, preventing memory leaks and simplifying component code.
- **Usage**: Place the `async` pipe at the end of the Observable expression in the template.

Javascript

```
// Component Class
myList$: Observable<any>;

// Template
<div *ngIf="myList$ | async as data">
  The latest value is: {{ data }}
</div>
```

15. What is the Purpose of `ngOnInit` Lifecycle Hook?

`ngOnInit()` is a lifecycle hook method that Angular calls **once** after it has created a component and performed all necessary dependency injection and data-bound property initialization (like `@Input()` properties).

- **Purpose**: It is the standard place to put initialization logic, such as:
  - Fetching initial data from a backend service.
  - Setting up component properties based on inputs.

16. What is the Angular CLI and What Are Its Benefits?

The **Angular CLI (Command-Line Interface)** is a command-line tool used to initialize, develop, scaffold, and maintain Angular applications. ✓

Benefits:

- **Scaffolding**: Quickly generates boilerplate code for components, services, modules, etc. (e.g., `ng generate component`). ✓
- **Consistency**: Enforces best practices and consistent structure across the project. ✓
- **Testing**: Built-in commands to run unit tests (`ng test`) and end-to-end tests (`ng e2e`). ✓
- **Building**: Compiles the application for development (`ng serve`) or production (`ng build --prod`). ✓
- **Upgrading**: Simplifies major framework updates (`ng update`). ✓

17. How Do You Generate a Component using Angular CLI?

You generate a component using the Angular CLI command `ng generate component` (or its short form `ng g c`), followed by the component name. ✓

Bash

```
# Full command
ng generate component user-profile

# Short form
ng g c user-profile
```

This command automatically creates the four necessary files (`.ts`, `.html`, `.css`, `.spec.ts`) and declares the new component in the nearest parent `NgModule`.

18. What is the Role of `main.ts` in an Angular App?

The `main.ts` file is the **entry point** for an Angular application. ✓

Its role is to bootstrap the application:

1. It imports necessary environment configuration.
2. It specifies which **Root Module** (`AppModule`) by default) should be used to start the application. ✓
3. It calls Angular's `platformBrowserDynamic().bootstrapModule(AppModule)` function to compile and launch the application within the browser.

19. What is the Purpose of `index.html` in Angular?

The `index.html` file is the **host page** that the user's browser loads when the application starts.

- It acts as the **single shell** for the entire Single-Page Application (SPA).
- It contains the necessary CSS links, scripts, and most importantly, the **root component selector** (usually `<app-root>`). The Angular application then takes control and renders all other components inside this single root tag. ✓

20. What is the Difference Between `@Input()` and `@Output()`?

Both `@Input()` and `@Output()` are decorators used for **Component Communication** in a parent-child relationship. ✓

Decorator	Role	Direction of Data Flow	Usage
`@Input()`	**Receiver/Input**	**Parent to Child**	Allows a parent component to pass data down to the child component via property binding.
`@Output()`	**Emitter/Output**	**Child to Parent**	Allows a child component to send data or events up to the parent component using an `EventEmitter`.

Export to Sheets

> 21. How do you pass data from parent to child component?
> 22. How do you emit events from child to parent.

Here are the answers to your questions about Angular component communication, services, and routing.

21. How Do You Pass Data From Parent to Child Component?

Data is passed from a parent component to a child component using the `@Input()` decorator.

1. **In the Child Component**: Declare a property decorated with `@Input()`. This marks the property as a public input property that can receive values from the outside.

   TypeScript

   ```
   import { Component, Input } from '@angular/core';

   @Component({ /* ... */ })
   export class ChildComponent {
     // Declare a property that can receive data from the parent
     @Input() myMessage: string;
   }
   ```

2. In the **Parent Component's Template**: Use **Property Binding** `[]` to bind the child component's input property to a parent component's property or expression.

   HTML

   ```
   <app-child [myMessage]="parentMessage"></app-child>
   ```

22. How Do You Emit Events From Child to Parent Component?

Events are emitted from a child component to a parent component using the `@Output()` decorator combined with the `EventEmitter` class.

1. **In the Child Component:**
   - Declare a property decorated with `@Output()`, initialized as a new `EventEmitter`.
   - Call the `.emit(data)` method on the emitter when the event occurs (e.g., inside a button click handler).

   TypeScript

   ```
   import { Component, Output, EventEmitter } from '@angular/core';

   @Component({ /* ... */ })
   export class ChildComponent {
     @Output() public dataRequested = new EventEmitter<string>();

     sendData() {
       this.dataRequested.emit('Data from child component.');
     }
   }
   ```

2. **In the Parent Component's Template:** Use **Event Binding** (`()`) to listen for the emitted event and execute a parent method, passing the emitted data (`$event`) as an argument.

   HTML

   ```
   <app-child (dataRequested)="handleChildData($event)"></app-child>
   ```

23. What is a Service in Angular?

A **Service** is a plain TypeScript class used to encapsulate **non-UI related logic** and data that can be shared across multiple components.

The core purpose of a service is to handle cross-cutting concerns, such as:

- Fetching data from a server (HTTP calls).
- Logging messages.
- Storing and sharing data between components.

A service is designated as injectable by using the `@Injectable()` decorator.

24. How Do You Create a Service using Angular CLI?

You create a service using the Angular CLI command: `ng generate service`. (or its short form `ng g s`), followed by the service name.

Bash

```
# Full command
ng generate service data-api

# Short form
ng g s data-api
```

This command creates the service file (`data-api.service.ts`) and automatically adds the `@Injectable()` provideIn: 'root' metadata, making the service a singleton available throughout the application.

25. What is Dependency Injection in Angular?

Dependency Injection (DI) is a core design pattern and mechanism in Angular where a class receives its dependencies (collaborator objects, typically **services**) from an external source rather than creating them itself.

- **How it works:** Angular's DI framework maintains a list of service providers (the **Injector**) and manages the creation and delivery of service instances upon request.
- **Benefit:** This makes components and services loosely coupled, easier to test (using mock dependencies), and more flexible.

TypeScript

```
// Angular injects the service instance into the constructor parameter
constructor(private logger:Service) { }
```

26. What is the Role of Providers in Angular?

Providers are instructions to the Angular **Dependency Injection (DI) system** on **how to create or obtain an instance of a dependency (a service)**.

- Providers are configured via the `providers` array in the `@NgModule` decorator, or more commonly using the `providedIn` property in the `@Injectable` decorator.
- `provideHttpClient` the modern default tells Angular to create a single, application-wide instance (a **singleton**) of the service in the root injector.

27. What is Routing in Angular?

Routing in Angular is the mechanism that allows a **Single-Page Application (SPA)** to navigate between different views (components) without requiring a full page reload from the server.

- The Angular Router interprets the browser's URL and determines which component to display in the dedicated area of the page.
- It supports features like navigation guards, lazy loading, and dynamic URL parameters.

28. How Do You Configure Routes in Angular?

Routes are configured by defining an array of route objects, typically in a file named `app-routing.module.ts`.

Each route object maps a URL **path** to a component.

TypeScript

```
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';

const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'about', component: AboutComponent },// Maps to /about
  { path: 'user/:id', component: UserDetailComponent },// Path with parameters
  { path: '**', component: NotFoundComponent },// Wildcard route for 404
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

29. What is a Router Outlet?

The **Router Outlet** (`<router-outlet>`) is a directive provided by the Angular Router.

- Purpose: It acts as a placeholder in the HTML template where the Angular Router inserts the component that corresponds to the current URL route.
- It is typically placed in the main application template (`app.component.html`) and marks the location where routed content is rendered.

HTML

```
<app-header></app-header>
<main>
  <router-outlet></router-outlet>
</main>
<app-footer></app-footer>
```

30. What is the Purpose of `RouterModule`?

The `RouterModule` is an Angular module that provides all the necessary services and directives for managing navigation within an application.

- **Key Functions:**
  1. Provides the `Router` service (for programmatic navigation).
  2. Provides the `RouterOutlet` directive (the component placeholder).
  3. Provides the `RouterLink` directive (for declarative navigation in templates, e.g., `<a routerLink="/about">`).
  4. Provides the static methods `forRoot()` (for the root module) and `forChild()` (for feature modules) to configure and initialize the routes.

> 31. What is a template-driven form?
> 32. What is a reactive form?
> 33. What is form validation in Angular?

Here are the answers to your questions about Angular forms, templates, and configuration files.

31. What Is a Template-Driven Form?

A **Template-Driven Form** is a simple approach to building forms in Angular where **most of the form logic, including validation and state tracking, is handled directly within the HTML template** using directives like `ngModel`.

- **Logic Location:** The source of truth for the form data resides in the HTML template.
- **Setup:** Requires importing the `FormsModule`.
- **Best For:** Simple forms, rapid prototyping, and applications where little complex validation is needed.

32. What Is a Reactive Form?

A **Reactive Form** is a model-driven approach where the **form structure, logic, and validation are defined programmatically** within the component's TypeScript class.

- **Logic Location:** The source of truth for the form data resides in the component class (an explicit object model).
- **Setup:** Requires importing the `ReactiveFormsModule`.
- **Best For:** Complex forms, dynamic forms, and applications requiring robust, unit-testable validation and dynamic manipulation.

33. What is Form Validation in Angular?

Form Validation in Angular is the process of ensuring that user input meets specific criteria before the data is processed or submitted.

Angular supports **both built-in validators** (like `required`, `minLength`, `pattern`) and **custom validators**. Validation is automatically managed by Angular's form directives, which track the form's state (e.g., `valid`, `invalid`, `dirty`, `touched`).

34. What is the Role of `FormControl`?

The `FormControl` class is the fundamental building block in **Reactive Forms**. It is an object that **tracks the value and validation status of a single input field** (form control) in the application.

- **Purpose:** To hold and manage the data for one form element, such as an input box or a checkbox.

35. What is the Role of `FormGroup`?

The `FormGroup` class is a collection of `FormControl` instances. It aggregates the values of each individual control into a single object and calculates the **overall status of the entire** form.

- **Purpose:** To represent the entire form as a cohesive unit, allowing you to manage the form data collectively (e.g., check `formGroup.valid`).

36. What Is `ngModel`: Used For?

The `ngModel` directive serves two main purposes:

1. **Template-Driven Forms:** It's the core directive that registers a form control with Angular, allowing the framework to track its value and validation state.
2. **Two-Way Data Binding:** When used with the "banana in a box" syntax (`[(ngModel)]`), it enables two way data binding on form input elements, synchronizing the view and the model.

37. What is the Difference Between `ngClass` and `ngStyle`?

Both are **Attribute Directives** used to dynamically manipulate the styling of HTML elements, but they target different properties:

Directive	Purpose	Input Type	Example
`ngClass`	**Dynamic CSS Classes.** Adds or removes CSS classes based on component data or conditions.	Takes a string, an array of strings, or an object (key-value pairs where the key is the class name and the value is a boolean condition).	`[ngClass]="{ 'is-invalid': isInvalid, 'form-control': true }"`
`ngStyle`	**Dynamic Inline Styles.** Applies inline CSS styles directly to an HTML element.	Takes an object where keys are camelCased CSS properties and their expressions/values.	`[ngStyle]="{ 'background-color': isActive ? 'green' : 'red', 'width.px': size }"`

Export to Sheets

38. What is the Purpose of `trackBy` in `ngFor`?

The `trackBy` function is an optional optimization feature used with the `*ngFor` structural directive.

- **Problem:** By default, when the data array changes (e.g., items are rearranged, added, or deleted), Angular re-renders **all** DOM elements corresponding to the list.
- **Solution:** The `trackBy` function returns a unique identifier for each item. When the array updates, Angular uses this ID to track which items have been added, removed, or moved, allowing it to manipulate only the necessary DOM elements.
- **Benefit:** This significantly improves rendering performance for large lists, especially when fetching updated data.

39. What Is the Role of `environment.ts`?

The `environment.ts` file, based its variables, the `environment.prod.ts` is used to **externalize environment-specific configuration variables**.

- Rule: Never configure environment-specific logic or change depending on the deploying environment (development, staging, production).
- **Content:** Typically includes variables like API base URLs, feature flags, and logging levels.
- **Usage:** The Angular CLI replaces the `environment.ts` file with the contents of `environment.prod.ts` when building for production (`ng build --configuration=production`), ensuring the correct configuration is bundled.

40. What is the Purpose of `angular.json`?

The `angular.json` file is the main **configuration file for the Angular CLI workspace**.

- **Purpose:** It defines the workspace structure, project settings, build configurations (e.g., assets, styles, scripts), testing settings, and default options for CLI commands.

It tells the CLI how to build, serve, and test each project within the workspace (an application can contain multiple projects).

41. What is the Role of `package.json`?

The `package.json` file is the standard configuration file for all Node.js/npm projects (including Angular).

- It holds **metadata about the project** and lists all its **dependencies** (packages needed to run or develop the application), scripts, `npm` commands, and project details.
- It dictates what Angular packages, libraries, and tools are used in the project.

42. What is the Purpose of `tsconfig.json`?

The `tsconfig.json` file is the configuration file for the **TypeScript compiler**.

- **Purpose:** It specifies the root files and compiler options required to compile the TypeScript code into JavaScript. This includes settings for the target JavaScript version, module resolution strategy, and where to find source files.

43. What is the Difference Between `dependencies` and `devDependencies`?

Both sections are found in `package.json`, but they serve different purposes:

Dependency Type	Purpose	When is Required?	Examples
`dependencies`	Packages required for the application to run in production.	At runtime in production and development.	`@angular/core`, `@angular/common`, `rxjs`, `zone.js`
`devDependencies`	Packages required only during development (e.g., testing, compiling, bundling).	Only during development (e.g., testing, compiling, bundling).	`@angular/cli`, `jasmine-core`, `karma`, `typescript`

Export to Sheets

44. What is the Role of `polyfills.ts`?

The `polyfills.ts` file contains **polyfill** scripts necessary to ensure the application runs correctly across a wide range of web browsers.

- **Rule:** Angular relies on modern JavaScript features that older browsers may not fully support. Polyfills add missing functionality to older browsers so they can execute the compiled application code.

45. What Is the Purpose of the `assets` Folder?

The `assets` folder is a standard directory used to hold **static files** that are copied directly into the application's build output (the deployment folder) without being processed by the Angular compiler or Webpack.

- **Purpose:** To store static content like images, fonts, external configuration files (e.g., JSON), or static HTML files that need to be accessible via a direct URL.

> 46. What is the role of styles.css?
> 47. What is the purpose of AppModule?
> 48. What is the default change detection strategy in Angular?
> 49. What is the role of zone.js?
> 50. What is the purpose of environment.ts?

Here are the answers to your final questions about Angular configuration and core mechanisms.

46. What is the Role of `styles.css`?

The `styles.css` (or `styles.scss`, etc.) file in the root of the Angular application is used to define **global styles** that apply across the entire application.

- **Rule:** It holds CSS rules that are not specific to any single component, such as:
  - Styles for the `<body>` element.
  - Font definitions and imports.
  - Global utility classes.
  - Styles for third-party libraries.
- Styles defined here are not subject to Angular's **View Encapsulation** and will cascade to all components unless overridden.

47. What is the Purpose of `AppModule`?

The `AppModule` (typically defined in `app.module.ts`) is the **root module** of a standard Angular application.

- **Purpose:** It serves as the primary entry point for the application's structure. Its `@NgModule` decorator tells Angular how to assemble the application. Specifically, it must:
  1. **Declare** the main application component (`AppComponent`).
  2. **Import** necessary framework modules (e.g., `BrowserModule`, `AppRoutingModule`).
  3. **Bootstrap** the root component, indicating which component Angular should load first when the app starts.

48. What is the Default Change Detection Strategy in Angular?

The default change detection strategy in Angular is `Default` (or `ChangeDetectionStrategy.Default`).

- **Mechanism:** With the default strategy, Angular performs a **full check** of every component in the component tree, from top to bottom, every time an asynchronous event occurs (such as a browser event, an HTTP response, a timeout, or a Promise resolution).
- **Alternative:** The `OnPush` strategy is a more performant alternative where a component is only checked if its `@Input()` properties change (or if an event is explicitly triggered within the component or checked manually).

49. What is the Role of `zone.js`?

`zone.js` is a library included in Angular that patches asynchronous browser APIs (like `setTimeout`, `addEventListener`, and XHR/fetch) to create an execution context called a "Zone".

- **Role:** It is crucial for Angular's **Change Detection**. Every time an asynchronous operation finishes, `zone.js` notifies Angular's wrapper zone (**NgZone**). This notification signals Angular that a change might have occurred, triggering the automatic change detection process across the application.
- It is the mechanism that automatically links asynchronous events to the Angular framework without the need for manual checks.

50. What is the Purpose of `platformBrowserDynamic()`?

The `platformBrowserDynamic()` function is a **part-in-time (JIT) compiler** tool used to launch the application in the browser during development.

- **Purpose:** When called, it first compiles the Angular components and modules in the browser before running the application. It returns a **platform object** that contains the function used to bootstrap (start) the root module.

TypeScript

```
// Inside main.ts:
platformBrowserDynamic().bootstrapModule(AppModule).catch(err => console.error(err));
```

- **Note:** For production builds, Angular uses **Ahead-of-Time (AOT)** compilation, which compiles the code offline during the build process, leading to smaller, faster applications. AOT compilation uses a different method to bootstrap the app.

> 1. What are Angular lifecycle hooks and how are they used?
> 2. How does Angular's change detection mechanism...

Here are the answers to your questions about advanced Angular concepts, including lifecycle hooks, change detection, routing, and RxJS.

1. What are Angular lifecycle hooks and how are they used?

Angular Lifecycle Hooks are methods provided by the Angular framework that allow you to tap into key events in the lifecycle of a component or directive, from creation to destruction. You do this them by implementing the corresponding interface (e.g., `OnInit`, `OnDestroy`) and defining the hook method in the component class.

Hook Method	Timing/Usage
`ngOnChanges`	Called before `ngOnInit` and whenever one or more data-bound input properties (`@Input()`) change.
`ngOnInit`	Called once, after Angular initializes the component's data-bound properties. **Best for initial fetching.**
`ngDoCheck`	Called every time the change detection cycle runs.

`ngAfterContentInit`	Called once after Angular projects external content into the component's view (i.e., content defined using `ng-content`).
`ngAfterContentChecked`	Called after the content has been checked projected into the component.
`ngAfterViewInit`	Called once after Angular initializes the regular template's view and its child views (called just after `ngAfterContentInit`). **Best for DOM manipulation**.
`ngAfterViewChecked`	Called after Angular checks the components view and its child views.
`ngOnDestroy`	Called just before Angular destroys the component. **Best for cleanup (unsubscribing, clearing intervals)**.

Export to Sheets

3. How does Angular's change detection mechanism work?

Angular's **change detection mechanism** is the process of detecting changes in application data and rendering those changes in the view (updating the DOM). 🔄

1. **Zone.js Notification**: The underlying library `zone.js` intercepts all asynchronous events (clicks, timers, HTTP calls). When an async operation completes, it notifies Angular via the **Signal**. 🔔
2. **Change Detection Trigger**: Angular starts its change detection cycle from the **root component** and traverses the entire **component tree**. 🌳
3. **Property Check**: For every component, Angular compares the **current value** of any property used in a template expression (e.g., `{{ myProperty }}`) with its **previous value**. 🔍
4. **DOM Update**: If the values differ, Angular updates the corresponding part of the DOM. ✨

By default, the strategy is `Default` (checks every component on every async event).

Export to Sheets

3. What is the difference between `ViewChild` and `ContentChild`?

Both decorators are used to query and access child elements, but they target different sources.

Decorator	Source of Element	Timing of Access	Primary Use Case
`@ViewChild()`	Elements defined within the component's **own template**.	Accessible after the `ngAfterViewInit` hook.	Accessing a child component, directive, or template reference defined directly in the component's view.
`@ContentChild()`	Elements projected into the component's view using `<ng-content>` (Content Projection/Transclusion).	Accessible after the `ngAfterContentInit` hook.	Accessing content passed **into** the component by its parent.

Export to Sheets

4. What is lazy loading in Angular?

Lazy Loading is a performance technique that invokes loading **feature modules** only when they are needed, typically when the user navigates to a specific route associated with that module.

- **Benefit**: It significantly reduces the initial bundle size of the application, leading to **faster startup times** for the user.

5. How do you implement lazy loading in Angular?

Lazy loading is implemented in the **routing configuration** of the application. 🗺️

1. Define the route using the `loadChildren` property instead of `component`. 📝
2. Use a **dynamic import** function to specify the path to the feature module file.

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

6. What is a resolver in Angular routing?

A **Resolver** (specifically a `Resolve` guard) is a class that implements the `Resolve` interface. It's used to fetch **necessary data before a route is fully activated and the component is initialized**. ⏳

- **Purpose**: It ensures that the component's data dependencies are met **before** the component is rendered. This prevents the view from flickering or showing empty states while waiting for an asynchronous API call to complete.

7. What is route guarding in Angular?

Route Guarding is the process of controlling navigation between routes. Guards are interfaces that the Angular router checks before or during navigation to decide if the navigation is allowed. 🚦 🔒

- **Purpose**: To enforce business rules, such as authentication (is the user logged in?), authorization (does the user have permission?), or data change confirmation (are there unsaved changes?).

8. What are `CanActivate` and `CanDeactivate` guards?

Guard Interface	Method	Purpose
`CanActivate`	`canActivate()`	Determines if a user can **access** (activate) a specific route. Used for authentication and authorization checks.
`CanDeactivate`	`canDeactivate()`	Determines if a user can **leave** (deactivate) the current route. Used to prompt users about unsaved changes.

Export to Sheets

9. What is the difference between `activatedRoute` and `Router`?

Class	Purpose	Scope
`ActivatedRoute`	Contains information about the **currently active route** associated with a loaded component.	**Local** to the component.
`Router`	The main **navigation service** used to programmatically control navigation within the application.	**Global** across the app.

Reading route parameters (`/profile/1`), query parameters (`?sort=price`), and static data (`data`).

Navigating to new routes programmatically (based on history / `Location`), and listening for navigation events.

Export to Sheets

10. How do you pass query parameters in Angular routing?

Query parameters (optional parameters appended to the URL after a question mark, e.g., `/products?sort=price`) are passed using the `queryParams` property of the `Router` method.

```
// 1. Programmatic Navigation (in a component)
this.router.navigate(['/products'], {
  queryParams: {
    category: 'electronics',
    page: 2
  }
});

// Resulting URL: /products?category=electronics&page=2

// 2. Template Using Routerlink:
<a [routerLink]="['/products']" [queryParams]="{ sort: 'name' }">View Products</a>
```

They are read in the target component using the `ActivatedRoute` service.

11. What is the role of `HttpClientModule`?

The `HttpClientModule` (from `@angular/common/http`) is an Angular module that provides the services for performing HTTP requests in Angular applications. 🌐

- **Role**: It introduces the `HttpClient` service, which offers features like:
  - Request/Response type checking.
  - Interceptors (for global request/response handling like adding tokens). 🔒
  - Streamlined error handling.
  - Observable-based API.

12. How do you make HTTP requests in Angular?

HTTP requests are made using the `HttpClient` service, which must be injected into the component or service. 📡

```
import { HttpClient } from '@angular/common/http';

// 1. Inject the service
constructor(private http: HttpClient) {}

// 2. Make the request (e.g., GET) and subscribe to the Observable
getUsers() {
  return this.http.get<User[]>('/api/users')
    .subscribe(data => {
      console.log(data);
    });
}
```

13. How do you handle errors in HTTP requests?

Errors in HTTP requests are handled by chaining the `.subscribe()` operator from RxJS onto the observable pipeline before subscribing.

1. The `.subscribe()` operator intercepts the error response. 🎣
2. Inside `.subscribe()`, you can perform logging, display a user message, and then return a new observable (e.g., an observable containing a default/fallback value).

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

getUsers() {
  return this.http.get<User[]>('/api/users').pipe(
    catchError(error => {
      console.error('An error occurred:', error);
      // Return an observable with a user-facing error or empty data
      return throwError(() => new Error('Something went wrong on the server.'));
    })
  );
}

// Component:
this.dataSvc.getUsers().subscribe(
  data => { /* success */ },
  error => { /* error message */ }
);
```

14. What is RxJS and how is it used in Angular?

RxJS (Reactive Extensions for JavaScript) is a library for composing asynchronous and event-based programs using **observables**.

- **Usage in Angular**: RxJS is fundamental to Angular for handling **asynchronous operations** in a more declarative way.
  - **HTTP**: `HttpClient` returns observables. 📡
  - **Forms**: Value changes on form controls are observables. 📝
  - **Routing**: Route parameter changes are observables. 🗺️
  - **State Management**: Events and state changes are often modeled as observables.

15. What are observables in Angular?

Observables are declarative objects that represent a stream of values over time. They are similar to Promises but handle **multiple, sequential values** (streams) rather than just a single future value. 🌊

- **Three Phases**:
  1. **Creation**: The observable is defined but does nothing.
  2. **Subscription**: The consumer (`.subscribe()`) initiates the stream. 🎣
  3. **Execution**: The observable delivers zero, one, or multiple values (`next`), an error (`error`), or a completion signal (`complete`).

16. What is the difference between Subject and BehaviorSubject?

Both are special types of Observables that can act as both an **Observer** (can receive data) and an **Observable** (can broadcast data). They are often used for event emitters or state management. 📢

Feature	`Subject`	`BehaviorSubject`
Initial Value	Does **not** require an initial value.	**Requires** an initial value in the constructor.
Late Value	Does **not** keep track of the last value emitted.	**Keeps track** of the latest value emitted.
New Subscriber	A new subscriber only receives values emitted **after** they subscribe.	A new subscriber immediately receives the **last emitted value** (or the initial value).

Export to Sheets

17. What is the role of `Subject`, `Observable` and `Promise`?

These are **Higher-Order Mapping Operators** used to handle scenarios where you receive an observable and need to perform an operation that returns another observable (e.g., making a second HTTP call based on the result of the first). They manage how the inner observable's stream is handled (flattening the inner observable into the outer one).

Operator	Concurrency	Order Guarantee	Use Case
`switchMap()`	Cancels previous inner observable.	Only cares about the latest material.	**Search/Autocomplete**. Discard old results if the user types quickly. **The most common choice**.
`mergeMap()`	Runs inner observables **concurrently**.	No (results come in as they arrive).	**Independent** tasks. Long-running, independent HTTP requests simultaneously.
`concatMap()`	Waits for the previous inner observable to complete before starting the next one.	**Yes** (results are guaranteed to be in order).	**Sequential Requests**. Operations that must happen in a strict order (e.g., Save → Log → Redirect).

Export to Sheets

18. How do you unsubscribe from observables?

Unsubscribing is critical to prevent **memory leaks**, especially with long-lived observables (like those from router events or intervals).

There are three primary ways:

1. **Manual Unsubscription (Old Way)**: Store the subscription and call `.unsubscribe()` in the `ngOnDestroy` hook. 🗑️

```
private subscription: Subscription;
ngOnInit() {
  this.subscription = myObservable.subscribe();
}
ngOnDestroy() {
  this.subscription.unsubscribe();
}
```

2. Using the `async` **Pipe (Best Way)**: The `async` pipe handles subscription and unsubscription automatically when the component is destroyed.
3. Using **RxJS Operators (Modern Way)**: Employing operators like `takeUntil` (for streams that complete after one value, like HTTP) or `takeWhile()` (for streams that run until a specific observable emits).

19. What is the role of `takeUntil` in Angular?

The `takeUntil` operator is used to elegantly handle unsubscription from long-lived observables. 🎯

- **Role**: It takes an observable as an argument. The source observable will continue emitting values until the argument observable **emits its first value**, at which point the source observable completes and automatically unsubscribes.

Implementation: You typically set up a `Subject` called `destroyS` in the component and call `this.destroyS.next(true)` inside `ngOnDestroy`.

```
private destroyS = new Subject<boolean>();

ngOnInit() {
  this.longRunningService.getData().pipe(
    takeUntil(this.destroyS), // This pipe stops when destroyS emits
    ).subscribe(data => { /* ... */ });
}

ngOnDestroy() {
  this.destroyS.next(!0); // Signal all streams using takeUntil to stop
  this.destroyS.complete();
}
```

20. What is the purpose of the `async` pipe with observables?

The `async` **pipe** is an impure pipe that simplifies working with observables in Angular templates.

- **Purpose**: To automatically **subscribe** to an Observable (or Promise) and extract the emitted value for display. When the component is destroyed, the pipe automatically handles the unsubscription to prevent memory leaks.
- **Syntax**: `{{ myObservable | async }}`
- **Benefit**: It allows the component logic to remain cleaner as manual subscription and unsubscription management in the component class is avoided.

21. What is the difference between template-driven and reactive forms?
22. How do you implement custom validators in Angular?

Here are the answers to your questions about advanced Angular concepts, including forms, change detection, rendering, and performance.

21. What is the difference between template-driven and reactive forms?

The key difference lies in the source of truth for the form's data model and control structure.

Feature	Template-Driven Forms	Reactive Forms
Model	**View/Template** (directives like `ngModel`)	**Component Class** (explicit `FormGroup`, `FormControl`)
Data Flow	Data flows through directives.	Explicit, imperative creation in TypeScript (`RxjsFormBuilder`).
Scalability	Better for **simple** or static forms.	Better for **complex**, large scale, or dynamic forms.
Testability	Lower, as logic is embedded in the templates.	**High**, as the model is independent and easily unit tested.
Immutability	Mutable (model changes can be tricky to track).	More manageable, explicit change tracking.

Export to Sheets

22. How do you implement custom validators in Angular?

Custom validators are implemented as functions that accept a `AbstractControl` argument (`control`) and return an object of validation errors if the validation fails, or `null` if it passes.

1. **Create the Validator Function**:

```
import { AbstractControl, ValidatorFn, ValidationErrors } from '@angular/forms';

// Factory function to accept a parameter (e.g., minlen arg)
export function customMinLengthValidator(minlen: number): ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null => {
    const bInstance = new DateLocaleUtil.getBufferInst();
    const currentLength = control.value?.length || 0;
    const age = calculateAge(control.value);

    return age >= minlen ? null : { 'minAge': { requiredAge: minlen, actualAge: age
  };
};
}
```

2. **Apply in Reactive Forms**: Use the function when defining the `FormControl`.

```
this.userForm = this.fb.group({
  dob: ['', minAgeValidator(18)] // Applying the custom validator
});
```

23. What is dynamic form generation?

Dynamic Form Generation is the process of building forms programmatically at runtime based on external data, typically a **JSON schema or configuration object**.

- **Purpose**: It's used when the form structure is not known at compile time or needs to change based on user roles, configurations, or external metadata.
- **Implementation**: Uses **Reactive Forms** heavily, often employing `FormGroup`, `FormArray`, and **structural directives** (`*ngFor`, `*ngSwitch`) in the template to iterate over the configuration object and render the appropriate input controls.

24. How do you use `FormBuilder` in reactive forms?

The `FormBuilder` is a helpful service that provides **syntactic sugar** for creating `FormGroup`, `FormArray`, and `FormControl` instances, reducing boilerplate code in Reactive Forms.

- **Inject** inject the `FormBuilder` service into the component's constructor.
- **Use** Use its methods (`group()`, `control()`, `array()`) to define the form structure and initial values/validators.

```
import { FormBuilder} from '@angular/forms';

constructor(private fb: FormBuilder) {}

ngOnInit() {
  this.profileForm = this.fb.group({
    // Value, then an array of validators
    firstName: ['', [Validators.required, Validators.minLength(2)]],
    lastName: ['',],
    address: this.fb.group({ // Nested FormGroup
      street: ['',]
    }),
    // Array
    phones: this.fb.array([this.fb.control('')]) // FormArray
  });
}
```

25. What is the role of `ChangeDetectorRef`?

The `ChangeDetectorRef` is a key interface that forms a **bridge** between the Angular Forms API (`FormControl`, `updateOn`) and a **native DOM element** or **custom form input component**.

- **Role**: It defines the methods needed to:
  - **Write values** from the Angular model to the DOM element (`writeValue()`).
  - **Propagate changes** from the DOM element back to the Angular model (`setValue()`).
  - **Propagate touch status** from the DOM element to the Angular model (`setTouched()`).
- It allows you to use `ngModel` or `FormControlName` on non-native input elements (like a custom color picker component).

26. What is the difference between `ngOnChanges()` and `ngDoCheck()`?

Both hooks relate to monitoring changes in a component, but differ in when and why they are triggered.

Hook	Triggered By	Usage	Frequency
`ngOnChanges()`	**Changes to `@Input()` properties only.** Triggered before `ngOnInit()` and whenever input properties change.	Used to react specifically to changes in data coming from parent.	Only when input properties change.
`ngDoCheck()`	**Every single change detection cycle,** immediately after `ngOnChanges()`.	Used to implement **custom change detection logic** to check for changes that Angular's default change detection might miss (e.g., changes within an object/array passed as an input).	Very frequently (on every browser/event cycle).

Export to Sheets

27. What is the role of `ngAfterViewInit()`?

`ngAfterViewInit()` is a lifecycle hook called **once** after Angular initializes the component's **view** and its **child views** (the component's own template and any components/elements queried using `@ViewChild()`).

- **Role:** It is the safe place to perform operations that rely on the view being fully rendered and accessible, such as:
  - Interacting with the native DOM using `ElementRef` or `Renderer2`.
  - Initializing third-party libraries that require DOM elements (e.g., a map widget).
  - Accessing children queried with `@ViewChild`.

28. What is the role of `ngAfterContentInit()`?

`ngAfterContentInit()` is a lifecycle hook called **once** after Angular initializes the **external content** that has been projected into the component's view using the `<ng-content>` element.

- **Role:** It is the safe place to perform operations that rely on the projected content being initialized and available, such as:
  - Accessing the projected content elements queried using `@ContentChild`.

29. What is the difference between `Renderer2` and direct DOM manipulation?

Method	Mechanism	Safety/Best Practice	When to Use
`Renderer2`	An **abstraction layer** that provides methods to safely manipulate the DOM.	**Recommended.** Abstracted from the underlying DOM, making it safe for environments outside of a browser (e.g., Server Side Rendering/SSR, Web Workers).	Almost always, when performing DOM manipulation.
Direct DOM Manipulation	Accessing and modifying the DOM using native browser APIs like `element.style.color = 'red'` or `document.getElementById()`.	**Discouraged.** Can cause errors in environments where rendering cycle, least privilege, or security are not guaranteed. Can bypass sanitization.	Rarely, and only when absolutely necessary and unavoidable.

Export to Sheets

30. What is the role of `ElementRef`?

`ElementRef` is a class that wraps the host element of a component or directive, providing a way to access the underlying native DOM element.

- **Role:** When injected, it allows low-level access to the element where the component or directive is applied.
- **Caution:** While it gives access to the native element (`elementRef.nativeElement`), its use should be minimized and combined with `Renderer2` for safe manipulation to maintain platform independence.

31. What is Angular Universal?

Angular Universal is the technology for performing Server-Side Rendering (SSR) for Angular applications.

- **Purpose:** It runs the initial application render on the **server** (Node.js environment) and generates static HTML content. This HTML is then sent to the browser, where the application fully bootstraps itself, taking over the static content to process called **hydration**.
- **Benefit:** Improves **SEO** (Search Engine Optimization) and perceived **performance** (Time to First Contentful Paint).

32. How do you implement server-side rendering in Angular?

Implementing SSR involves setting up the Angular Universal environment, typically using the Angular CLI's dedicated command:

1. **Add Universal:** Run `ng add @nguniversal/express-server`. This sets up the necessary files:
   - `server.ts` (the Node.js server file).
   - `app.server.module.ts` (the module compiled for the server).
2. **Build:** The application is built twice: once for the client (browser) and once for the server.
3. **Run:** The Node.js server runs to render requests using the Angular application using the server bundle, and sends the resulting HTML to the client.

33. What is Ahead-of-Time (AOT) compilation?

Ahead-of-Time (AOT) compilation is the default compilation strategy used by Angular for production builds.

- **Mechanism:** The Angular components and templates are compiled into efficient JavaScript code **during the build process** (before the browser downloads and runs the code).
- **Benefit:**
  1. **Faster Rendering:** The browser receives already compiled code, leading to faster application startup.
  2. **Smaller Bundles:** The Angular compiler and template parsing logic are not needed at runtime.
  3. **Error Detection:** Template errors are caught during the build phase, not at runtime.

34. What is `zone.js` in Angular?

`zone.js` is the name of Angular's **modern-rendering engine and compilation pipeline**, introduced fully in Angular 9.

- **Purpose:** It was the initial application render on the **server** (Node.js environment) and generates static HTML content. This HTML is then sent to the browser, where the application fully bootstraps itself, taking over the static content to process called **hydration**.
- **Benefits:** Improves **SEO** (Search Engine Optimization) and perceived **performance** (Time to First Contentful Paint).

35. What is Angular Signals?

Angular Signals are a new system introduced in v16+ for fine-grained **change detection** and state management.

- **Concept:** They are wrapper objects for reactive values. When a signal's value changes, Angular only checks and updates the components/templates that directly depend on that specific signal, rather than running a full check of the component tree.
- **Role:** They enable **local, deterministic, and synchronous** reactivity, potentially moving Angular away from the reliance on `zone.js` and the default global change detection strategy.

36. What is the Role of Injector?

The **injector** is the core component of Angular's **Dependency Injection (DI)** system.

- **Role:** It is a runtime mechanism that stores a **token** (usually a class type like `LoggerService`) for a **provider** (the recipe for creating that service).
- **Hierarchy:** Angular maintains a hierarchical tree of injectors (one root injector, and potentially others at the component/module level). When a component requests a dependency, the injector attempts to resolve it, searching up the tree until it finds a provider.

37. What Is a multi-provider In Angular?

A **Multi-Provider** (enabled by setting `multi:true` in the provider configuration) is a feature that allows multiple providers to be registered for a single injection token.

- **Role:** Instead of receiving a single instance of a service, the component receives an **array of all registered services** or values for that token.
- **Use Case:** Used for extension points, such as:
  - Registering multiple `HTTP_INTERCEPTORS` (all interceptors are collected into an array and run sequentially).
  - Registering multiple `APP_INITIALIZER` functions.

38. What is the Role of `APP_INITIALIZER`?

The `APP_INITIALIZER` is an injection token used to provide one or more functions that Angular must execute **before the application starts rendering** (before the root component is bootstrapped).

- **Role:** It allows you to perform critical setup tasks, typically asynchronous ones, such as:
  - Loading configuration settings from an API.
  - Initializing third-party libraries.
  - Performing runtime feature flag checks.
- **Mechanism:** The functions provided must return a `Promise`. Angular waits for all promises to resolve before proceeding with application startup.

39. How do you create a custom directive?

A custom **Attribute Directive** is created to add specific behavior or attributes to existing elements.

1. **Generate:** Use the CLI: `ng generate directive highlight`.
2. **Define:** Decorate a class with `@Directive` and provide a CSS selector.
3. **Implement Input:** Use `@Input()` and/or `@HostBinding` to manage the state and attributes.
4. **Host Listener:** Use `@HostListener` to react to events on the host element (e.g., mouse enter/leave).
5. **Host Binding:** Use `@HostBinding` to dynamically set properties on the host element (e.g., `style.color`).

TypeScript

```
@Directive({
  selector: 'appHighlight' // Selector used in templates
})
export class HighlightDirective {
  constructor(private el: ElementRef, private renderer: Renderer2) {}

  @HostListener('mouseenter') onMouseEnter() {
    this.renderer.setStyle(this.el.nativeElement, 'background-color', 'yellow');
  }
}
```

40. How do you create a custom pipe?

A custom **Pipe** is created to transform data for display in templates.

1. **Generate:** Use the CLI: `ng generate pipe reverse`.
2. **Define:** Decorate a class with `@Pipe` and provide a unique `name`.
3. **Implement:** The class must implement the `PipeTransform` interface and define the `transform` method, which contains the transformation logic.

TypeScript

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'reverse' // Pipe name used in templates
})
export class ReversePipe implements PipeTransform {

  transform(value: string): string {
    if (!value) return '';
    return value.split('').reverse().join('');
  }
}
```

AI: What is the role of signals?
AI: What is the role of ChangeDetectorRef?
AI: What is the difference between OnPush and Defaul...

Here are the answers to your questions about advanced Angular concepts, including change detection, state management, and testing.

41. What is the role of NgZone?

The **NgZone** is an Angular service that wraps the open-source `zone.js` library. Its primary role is to create an execution context (a "zone") where Angular's asynchronous operations run.

- **Trigger Change Detection:** When an asynchronous operation finishes within **NgZone** (e.g., a button click, an HTTP response, a `setTimeout`), the NgZone automatically notifies Angular that the application state might have changed. This notification **triggers Angular's change detection mechanism**, which then updates the view.
- **Performance Optimization:** It allows developers to run certain intensive or frequent asynchronous tasks outside of Angular's change detection. This prevents the task from unnecessarily triggering change detection, improving performance.

42. What is the role of ChangeDetectorRef?

The `ChangeDetectorRef` is a class that allows developers to **programmatically control and interact** with the component's change detector. It is used to fine-tune change detection behavior.

- **Key Methods:**
  - `markForCheck()`: Used with the `OnPush` strategy. It tells Angular to check the current component during the next change detection cycle, even if its inputs haven't changed.
  - `detectChanges()`: Forces an **immediate, one-time check** of the component and its children, regardless of the change detection strategy.
  - `detach()`: Stops the change detector for the component and its children (useful for highly optimized or immutable views).

43. What is the difference between OnPush and Default change detection?

These are the two main strategies that decide when Angular checks a component for changes.

Strategy	Trigger	Checks	When to Use
`Default` (Checks Always)	Any asynchronous event (click, timer, HTTP).	Checks all components in the tree, regardless of whether data has changed or not.	Simple apps, high frequency data changes.
`OnPush`	**Limited checks.** Only runs change detection when: 1. **Input reference** changes (`@Input()`). 2. **Event** originates from the component. 3. Change detection is manually triggered via `ChangeDetectorRef`.	Highly optimized. Skips entire subtrees if inputs are unchanged. Requires working with immutable data structures.	Large, complex applications where performance is critical.

Export to Sheets

44. What is the role of Ngrx in Angular?

Ngrx is a set of libraries inspired by the Redux pattern, designed for **state management** in Angular applications.

- **Role:** It provides a **single source of truth** (the Store) for the application's state and enforces a unidirectional data flow (UDF) to manage state changes predictably and transparently. This makes complex applications easier to understand, debug, and test.
- **Core Principle:** State changes are only allowed through explicit **Actions**, ensuring that data changes are transparent and centralized.

45. What are Actions, reducers, and effects in NgRx?

These are the three fundamental concepts that define the flow of data in the NgRx architecture:

1. **Actions (Declarative Intention):** to change the state in pure functions. They are dispatched by components to describe what happened (e.g., `[Products Page] Load Products`). They carry a unique type and an optional payload.
2. **Reducers:** Pure functions responsible for handling state transitions. They take the current state and an action as inputs and return a new, immutable state. Reducers must not perform asynchronous operations or modify the old state.
3. **Effects:** Handles the **side effects** (operations that interact with the outside world). They listen for dispatched actions and perform tasks like making HTTP calls, handling routing, or interacting with local storage. Effects dispatch new actions upon completion (e.g., `[Products API] Load Products Success`).

46. What is the role of Store in NgRx?

The **Store** is the single, centralized state container in an NgRx application.

- **Role:**
  - Holds the complete state tree of the entire application.
  - Provides methods to `dispatch` actions.
  - Provides methods to `select` pieces of state as Observables.
- **Mechanism:** The Store is the single entity that calls the reducers when an action is dispatched, updating the state immutably. Components subscribe to the Store to receive state updates.

47. How do you test Angular components?

Angular components are typically tested in two ways:

1. **Isolated Unit Testing:** Testing the component's **TypeScript class** in isolation, without rendering its template. This is done by simply instantiating the class and mocking any service dependencies.
2. **Shallow/Deep Unit Testing (Component Testing):** Testing the component's **interaction with its template (View).** This is done using the `TestBed` utility, which creates a testing module environment and renders the component.

The focus is on verifying that user events trigger the correct methods, inputs are rendered correctly, and outputs emit the expected data.

48. What is the role of TestBed?

The `TestBed` is the primary utility class in Angular's testing module (`@angular/core/testing`).

- **Role:** It configures and initializes a **testing module** environment (analogous to `@NgModule`) where the component, service, or module can be tested.
- **Functions:** It allows you to:
  - `configureTestingModule()`: Declare components, modules, and providers needed for the test.
  - `createComponent()`: Create an instance of the component to be tested and its associated fixture.
  - `inject()`: Retrieve services from the testing injector.

49. How do you mock services in unit tests?

Services are mocked in unit tests to isolate the component being tested from its dependencies, ensuring that the test focuses only on the component's logic.

There are two common ways to mock services using `TestBed`:

1. **Using a Mock Class/Object:** Define a simple class or object that mimics the required methods of the real service.

TypeScript

```
// Mock service
const mockUserService = {
  getUsers: () => of([{ id: 1, name: 'Test User' }]),
  isAdmin: () => false
};

TestBed.configureTestingModule({
  providers:
    // Provide the mock instead of the real service
    { provide: UserService, useValue: mockUserService }
  ]
});
```

2. **Using Mock/Stubbed Models:** Use a mocking library to create a dynamic mock object, which is then provided to the TestBed using `useValue`.

50. What is the difference between unit testing and integration testing in Angular?

Test Type	Scope	Goal	Dependencies	Tooling
Unit Testing	A small, isolated unit of code (a function, one service, method, component class).	To verify that a specific unit of code works as expected.	Dependencies are **mocked or stubbed** to isolate the unit.	Karma/Jasmine, Jest/Mocha, TestBed.
Integration Testing	A group of units (e.g., Component + Service, Module + Router) or a full feature flow (e.g., API call, data transformation, UI update).	To verify that different parts of the application **work correctly together.**	Uses real services or dependencies (or a test version of them) to test the interactions.	Protractor/Cypress (E2E) or TestBed (for component integration).

Export to Sheets

Analysis and I Have

The content you uploaded appears to be an image of a list of questions covering **Java Swift**, Array/List, HashMap, and SOP concepts, during Reporting Data, JPA, and Angular.

Since the image content is not directly accessible, I will transcribe and provide the solutions for the questions based on the typical structure of such interview question lists.

Part 1: Java & Object-Oriented Programming

	Question	Solution
1.	**Interface vs Abstract Class.**	**An Interface** can only contain method signatures (pre-Java 8?), constant fields, and no state or implementation details (pre-Java 8?) Actually, it does can implement multiple interfaces. **An Abstract Class** can contain abstract methods, concrete methods, constructors, and state (instance variables).
2.	**Getters and Setters**	Methods used to enforce **encapsulation**. **Getters (accessors)**, typically prefixed with `get()`, return the value of a private field. **Setters (mutators)**, typically prefixed with `set()`, modify the value of a private field, allowing for validation before assignment.

5.	**Code for `Array(List)`**	`Array(List)` is a dynamic array implementation of the `List` interface. It stores elements in the order they are added and allows duplicates. **Example:** `Array(Int) numbers = Array(Int).new(3, 0); numbers[0] = 10; numbers[1] = 20; numbers[2] = 30; puts numbers.inspect` `numbers.delete(20); puts" new: #{numbers.inspect}" puts" #{numbers[0]}, #{numbers[1]}, #{numbers[2]}" Output: [10, 0, 0] new: [10, 0] 10, 0, 0`
6.	**Code for sorting custom objects in `Array(Object)`**	You simply use the class name as the generic type parameter. **Example:** `class Person; property name : String; property age : Int32; def initialize(name : String, age : Int32); @name = name; @age = age; end; end; people = [ Person.new("Alice", 30), Person.new("Bob", 25), Person.new("Charlie", 35) ] puts" Accessing the second object's properties" `
7.	**`HashMap` in Java**	`HashMap` is a core of Java's Collection Framework that implements the `Map` interface. It stores data in **key-value pairs**. It provides fast lookup time (average O(1)) and does not maintain any specific order of elements. It facilitates the retrieving value for that key. Elements are stored in no particular order.

Export to Sheets

Part II: Spring Boot & Spring Data JPA

	Question	Explanation
8.	**Loose Coupling in Spring Boot, Spring in Spring Boot**	**Tight Coupling** means one class highly dependent on the concrete implementation of another, making code fragile and difficult to change/test. **Loose Coupling** achieves flexibility by using interfaces. **Dependency Injection (DI)** and **Inversion of Control (IoC)** frameworks do not hardcode dependencies, instead, the framework injects them, allowing implementations to be swapped easily.
9.	**Spring Boot Auto-Configuration**	Spring Boot's automatic configuration is a mechanism that automatically configures components (beans) based on the libraries present on the classpath and defined properties. It simplifies setup by providing sensible defaults. Developers can override these defaults if needed.
10.	`pom.xml` / `gradle.properties` / `application.properties`	These three elements form the **Maven/Gradle** ecosystem, which uniquely identify a dependency artifact. The `groupId` identifies the project group (e.g., `org.springframework`), the `artifactId` specifies the specific library (e.g., `spring-boot-starter-web`), and the `version` is the specific release (e.g., `2.7.0`).
11.	**Spring Boot Required Components**	The key components often cited are **Starters** (pre-configured dependency sets like `spring-boot-starter-web`), **Auto-Configuration** (the brain behind setup), **IoC Container** (manages bean lifecycle), and **Actuator** (for monitoring/management). Other important components include `@SpringBootApplication` (enables core features), `@Configuration`, `@ComponentScan`, and `application.properties`.
12.	**Spring Data JPA Repository**	Spring Data JPA allows you to define repository methods without writing implementation code. Simply extending `JpaRepository` provides access to common CRUD operations. For custom queries, use `@Query` annotation with JPQL or native SQL syntax. Method naming conventions (`findBy...`, `countBy...`, `existsBy...`) also enable automatic query generation.

Export to Sheets

Part III: Angular & Front-End Concepts

	Question	Solution
13.	**Angular Lifecycle Hooks**	Methods that allow you to tap into key events in a component's lifecycle (creation, change detection, and destruction). The common sequence is `ngOnChanges` (when input properties change), `ngOnInit` (initialization/setup), `ngDoCheck` (custom check), `ngAfterContentInit`/`ngAfterContentChecked` (projected content), `ngAfterViewInit`/`ngAfterViewChecked` (component's view), `ngOnDestroy` (cleanup).
14.	`package.json`	The configuration file for any Node.js project. It lists project **metadata** (name/version), **scripts** for running/building, and specifies all project **dependencies** (libraries/packages) and **devDependencies** (tools needed only for development tools).
15.	`app.ts` file in **Angular**	In the first useless **add tests** for the corresponding Type/Service file, `app.component.spec.ts` tests `app.component.ts`. The `.spec` convention signifies unit tests. These files are typically run by a testing framework and **Karma** (the test runner) to execute these tests.
16.	`HttpClient` in **Angular**	The `HttpClient` service (from `@angular/common/http`) is used to make **HTTP requests** (GET, POST, PUT, DELETE) to backend servers. It returns **RxJS Observables**, which handle asynchronous data streams. The component subscribes to these observables to receive data and update the UI. Error handling is typically done using RxJS operators like `catchError`.
17.	**API Call - Service - Component**	**Create a Service** like `product.service.ts` in a dedicated folder to handle HTTP requests. **Inject** the service into the component using Angular's dependency injection. **Call the service** method in the component (often `ngOnInit`) and `.subscribe()` to the returned Observable to process the data or handle errors.
18.	**API Call - Component - Service** (alternative order)	This typically refers to passing data or events between components after an initial API call and service setup. Data flows from parent to child via `@Input()` properties, and from child to parent via `@Output()` properties. **Child-to-Parent (CUP):** The child component emits an event using `@Output()` and `EventEmitter`, the parent listens to this event and stores the received data in its own `@Input()` property and pass it to other children.
19.	**Angular Providers**	Provides root instructions to the Dependency Injection (DI) system on how **to create or obtain an instance of a dependency**. It tells Angular what value, or factory, they are configured in the `providers` array of an `@NgModule` or `@Component`. Providers can be classes, values, or factory functions, allowing for flexible dependency management and testing.
20.	**Nginx (General)**	Nginx is a **static management library** for Angular based on the Redux pattern. It establishes a **single source of truth** (the Store) and a **unidirectional data flow** (UDF) to manage application state predictably. Components dispatch actions to request changes, ensuring state updates are centralized and traceable.
21.	**Angular Signals (General)**	A modern feature introduced in v16+ for fine-grained reactivity and state management. Signals are wrappers around values that notify interested consumers when their value changes. Angular can then efficiently update only the change detection only on the specific parts of the UI/DM that depend on the changed signal, offering a path to better performance and reduced reliance on `zone.js`.
22.	**Angular Guards (General)**	This term can refer to two things: 1. `@GuardsCheck` / `@GuardsCheckStart` / `@GuardsCheckEnd` are lifecycle hooks that run during the router's navigation process. 2. Guards are third-party state management library focused specifically on simplifying server-side data fetching, caching, synchronization, and automatic re-fetching of server data.

Export to Sheets

give me detailed answer for the all questions

The comprehensive response covers all the questions from your list spanning **Java, Spring Boot, and Angular**, providing detailed answers for each section.

Part I: Java and Object-Oriented Programming (OOP)

Feature	**Interface (since 8)**	**Abstract Class**
Type	Defines a contract (what a class must do)	Provides a template (what a class is or can do)
Implementation	Can only contain **abstract methods** (as one is a **default/static methods** (since 8))	Can contain **abstract methods** (no body) and **concrete methods** (with implementation)
Body (Implementation)	Only public static final variables	Can have any variables
Inheritance	A class can **implement** multiple interfaces (solves diamond problem)	A class can **extend** only one abstract class
Constructor	Cannot have a constructor	Can have a constructor (called by subclasses)

2. Options and Scenarios

`Optional<T>` (`Optional<String>`, `Optional<Integer>`, `Optional<Boolean>`, `Optional<Double>`) `Long.valueOf(long)` used to enforce **encapsulation** by controlling external access to an object's private fields.

- **Getter:** Returns the value of a private field.
- **Setter:** Allows external code to change the value of a private field, often including **validation logic** before the value is assigned. This prevents direct, uncontrolled modification of fields.

3. Code for `Array(List)`

An `Array(List)` is a resizable array implementation of the `List` interface.

```
Array(List)
```

```
import java.util.Arrays;

public class ListExample {
    public static void main(String[] args) {
        // Declaration and initialization
        Array(List<String> fruits = new ArrayList<>();

        // Add elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");
        System.out.println("Initial fruits: " + fruits);

        // Access elements
        String firstFruit = fruits.get(0);
        System.out.println("First fruit: " + firstFruit);

        // Remove elements
        fruits.remove("Banana");
        System.out.println("After removing Banana: " + fruits);

        // Size
        int size = fruits.size();
        System.out.println("Size: " + size);

        // Iterate through the array
        for (String fruit : fruits) {
            System.out.println("Fruit: " + fruit);
        }
    }
}
```

4. Code for Storing Custom Objects in `Array(List)`

Custom objects are stored by using the class name as the generic type parameter.

```
Array(List<Product>
```

```
import java.util.Arrays;

class Product {
    String name;
    double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }
}

public class CustomObjectExample {
    public static void main(String[] args) {
        Array(List<Product> products = new ArrayList<>();

        // Adding custom objects
        products.add(new Product("Laptop", 1200.00));
        products.add(new Product("Mouse", 25.00));

        // Accessing and using object properties
        String firstProduct = products.get(0).getName();
        System.out.println("First item: " + firstProduct);
    }
}
```

5. `HashMap` in Java

`HashMap` is a key component of the Java Collections Framework that implements the `Map` interface.

- It stores data as **key-value pairs**.
- **Keys must be unique:** attempting to insert a duplicate key replaces the existing value.
- It provides fast lookup, insertion, and deletion (O(1) time complexity for common operations like `put()`, `get()`, and `remove()`).
- It does not maintain any specific order of elements.
- It's **unsynchronized** (insertion order is not preserved).

Part II: Spring Boot and Spring Data JPA

6. Loose Coupling vs. Tight Coupling in Spring

- Right Coupling: Provides what it does directly depends on the concrete implementation of another class.
- Loose Coupling: Achieves flexibility by using interfaces. The dependency implementation can be swapped without changing the consuming class.

```
Java
```

```
// Tight Coupling: MyService directly creates and depends on ConcreteDatabase
class MyService {
    private ConcreteDatabase db = new ConcreteDatabase();
}

// Loose Coupling: MyService depends on the abstraction IDatabase
class MyService {
    // Dependency injected by Spring Boot/Spring Framework
    private IDatabase db;
}

interface IDatabase {
    void save(String data);
}

class ConcreteDatabase implements IDatabase {
    @Override
    public void save(String data) {
        // ...
    }
}
```

7. Spring Boot Auto-Configuration

Auto-Configuration is a mechanism that automatically configures the Spring application context based on the JAR dependencies present in the classpath.

- **Goal:** To reduce the amount of manual configuration (XML or `@Configuration` classes) required for common Spring features.
- **Mechanism:** When the application starts, Spring Boot looks for specific classes and annotations using conditional logic (e.g., `@ConditionalOnClass`). If a library (like Hibernate or Tomcat) is detected, Spring Boot automatically creates and configures the necessary beans (e.g., `DataSource`, `EntityManager`, `EmbeddedServlet`).
- **Example:** Adding `spring-boot-starter-web` automatically configures an embedded Tomcat server and Spring MVC.

8. `pom.xml` / `gradle.properties` / `application.properties`

These three files uniquely identify a dependency artifact, which uniquely identify any project within Maven or Gradle ecosystems, and it is defined as follows:

- `groupId`: Uniquely identifies the project group or organization (e.g., `org.springframework`). It is usually follows Java package naming conventions (reversed domain name).
- `artifactId`: The **name** of the artifact itself (the library or module), often corresponding to the JAR file name (e.g., `spring-boot-starter-web`).
- `version`: The specific release number of the artifact (e.g., `2.2.0.BUILD-SNAPSHOT`). Versions are used for ongoing development builds.

9. Spring Boot Required Components

The four main components that underpin the Spring Boot ecosystem are:

1. **Spring Boot Starters:** Pre-packaged dependency sets (e.g., `spring-boot-starter-web`) that aggregate common or related dependencies into a single, easy-to-manage entry.
2. **Spring Boot Auto-Configuration:** The core mechanism that automatically configures the application based on classpath libraries.
3. Spring Boot CLI (Command Line Interface): A tool for quickly prototyping Spring applications using Groovy scripts.
4. **Spring Boot Actuator:** Provides production-ready features like monitoring, metrics, and health checks via HTTP endpoints.

The core of a typical application is the `@SpringBootApplication` annotation, which is a convenience wrapper for `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`.

10. Spring Data JPA Queries

Spring Data JPA allows developers to define data access logic concisely via repository interfaces, supporting queries in three primary ways:

1. **Derived Query Methods (Method Naming Convention):** Spring automatically generates the query logic based on the method name.
   - Example: `findByLastnameAndFirstnameIsNot(String lastname, String nameIs)`
2. **JPQL (Java Persistence Query Language) Queries:** Using the `@Query` annotation to write queries that operate on **entities and their fields** rather than database tables/columns.
   - Example: `@Query("select u from User u where u.emailAddress = ?1")`
3. **Native SQL:** Using the `@Query(nativeQuery = true)` annotation to write raw, database-specific SQL.

Part III: Angular and Front-End Concepts

11. Angular Lifecycle Hooks

These are interface methods that allow a developer to execute code at specific points in a component's or directive's life, from creation to destruction.

`ngOnChanges`	Runs when input properties change.
`ngOnInit`	Runs once, typically used for initialization.
`ngDoCheck`	Runs after every change detection cycle.
`ngAfterContentInit`	Runs after content (projected content) has been initialized.
`ngAfterContentChecked`	Runs after every check of projected content.
`ngAfterViewInit`	Runs after the component's view has been initialized.
`ngAfterViewChecked`	Runs after every check of the component's view.
`ngOnDestroy`	Runs just before the component is destroyed. **Crucial for cleanup** (unsubscribing from observables, clearing timers, etc.) to prevent memory leaks.

Export to Sheets

12. `package.json`

The configuration file for any Node.js project. It lists project **metadata** (name/version), **scripts** for running/building, and specifies all project **dependencies** (libraries/packages) and **devDependencies** (tools needed only for development tools).

13. `app.ts` file in **Angular**

In the first useless **add tests** for the corresponding Type/Service file, `app.component.spec.ts` tests `app.component.ts`. The `.spec` convention signifies unit tests. These files are typically run by a testing framework and **Karma** (the test runner) to execute these tests.

14. `HttpClient` in **Angular**

The `HttpClient` service (from the `@angular/common/http` module) is Angular's primary mechanism for making HTTP requests to a backend API.

- It uses RxJS Observables to manage the asynchronous nature of requests.
- **Usage:** A service injects `HttpClient` and calls methods like `get()`, `post()`, `put()`, `delete()`, returning an Observable. Components or other services then subscribe to the Observable to receive the response data or handle errors.
- **Benefits:** Supports request/response interceptors (for adding headers or logging) and provides robust, type-checked data handling.

15. API Call - Service - Component

API calls should be encapsulated in an injectable **Service** to adhere to the separation of concerns.

1. **Service:** Defines a method (e.g., `fetchProducts()`) that calls the API and returns `HttpClient` Observable(s).
2. **Component:** Injects the custom service. Calls the service method, then calls `.subscribe()` on the returned Observable, typically in the `ngOnInit` hook, to process the data and assign it to a component property for display.

16. API Call - Component (PYC) and CUP

```
Component -> Service -> Backend -> Service -> Component
```

Parent Component	`@Input()` property passes data down	Parent passes the API call (how data is received, the parent passes that data down to the child component using property binding `[child-input]="parent-data"`)
Child Component	`@Output()` property emits events up	Child handles the API call (how the call is made, the child uses `@Output()` and `EventEmitter` to send the data back to the parent component. `this.myOutputProperty.emit(data);` `[child-output]="handleChildEvent($event)"`)

Export to Sheets

17. Angular Providers

A **Provider** is an instruction to the **Dependency Injection (DI)** system on how to acquire an instance of a dependency (a service, value, or function) associated with a given token.

- They are defined in the `providers` array of an `@NgModule` or `@Component`.
- `providers: [ { provide: MyService, useClass: MyMockService } ]` registers the service as an application-wide **singleton**.
- Providers ensure loose coupling by mapping an abstract token to a concrete implementation (e.g., `useClass`, `useValue`, `useFactory`).

18. Nginx (General)

Nginx is a pattern and set of libraries for managing **global application state** in a controlled, predictable manner, based on the **Redux** pattern.

- Its goal is to establish a **Single Source of Truth** (Store) and enforce a **Unidirectional Data Flow (UDF)**.
- **Core Concepts/ Actions:** Immutable to change state) — **Reducers:** Pure functions that handle state changes — **Store:** Holds state — **Selectors:** Used select — **Effects:** Handle side effects like API calls.

19. Angular Signals (General)

Angular Signals are a modern reactivity primitive for defining values that notify interested consumers when their value changes.

- **Role:** They enable **fine-grained change detection** by making the framework aware of exactly which components and templates depend on which piece of state. This allows Angular to update the DOM very precisely without running the entire change detection cycle (eventually leading to the removal of `zone.js`).
- **Benefit:** Improved performance and a simpler, more explicit mental model for reactivity.

20. Angular Guards (General)

The term "Angular Guards" usually refers to two specific developers used for accessing child elements:

- `@GuardsCall` / `@GuardsCallStart` / `@GuardsCallEnd`: Used to query for elements **within the component's own template** (the host view).
- `@GuardsContent` / `@GuardsContentStart` / `@GuardsContentEnd`: Used to query for elements that are **projected into the component** from its parent (e.g., `<ng-content>`).

These queries grant access to component instances, directives, or native DOM elements (e.g., `ElementRef`).

any do we need interface? if there is abstract class , and also explain why does it provide 100% abstraction
i.e. explain difference between abstract class and interface

That's an excellent set of questions that goes right to the heart of Object-Oriented Programming (OOP) principles and design patterns. While an abstract class can provide 100% abstraction, interfaces are still crucial for achieving flexibility, multiple inheritance, and decoupling in modern software development.

Here's a detailed explanation of why we need interfaces even when abstract classes exist, and why abstraction alone isn't enough:

Why Do We Need Interfaces Even When Abstract Classes Exist?

While abstract classes provide a template and can enforce 100% abstraction, interfaces offer a different kind of abstraction and solve problems that abstract classes cannot address:

1. **Multiple Inheritance of Type:**
   - **Abstract Class:** A class can only inherit from one abstract class (in most languages like Java, C#, PHP).
   - **Interface:** A class can implement multiple interfaces. This allows a class to fulfill multiple contracts and behave like different types of objects.
2. **Decoupling and Loose Coupling:**
   - **Abstract Class:** Creates a tight coupling between the parent class and its subclasses. The subclasses are directly dependent on the implementation details of the abstract class.
   - **Interface:** Promotes loose coupling. Classes that implement an interface only depend on the contract (the method signatures), not on the specific implementation details of any other class. This makes it easier to change implementations without affecting dependent code.
3. **Defining Capabilities (Mixin Behavior):**
   - **Abstract Class:** Defines what an object is (e.g., `Animal`, `Vehicle`).
   - **Interface:** Defines what an object can do (e.g., `Flyable`, `Swimmable`, `Serializable`). These capabilities can be added to unrelated classes.
4. **Contract Enforcement:**
   - **Abstract Class:** Enforces implementation of abstract methods by its subclasses.
   - **Interface:** Enforces that any class implementing it must provide implementations for all its methods, ensuring a specific contract is fulfilled.

Why Abstraction Alone Isn't Enough

Abstraction hides complexity by showing only essential details. However, relying solely on abstract classes for abstraction limits flexibility and creates tight coupling. Interfaces provide a more flexible and decoupled way to achieve abstraction by defining contracts that classes can implement regardless of their inheritance hierarchy.

Example: Abstract Class vs. Interface

Consider a system for different types of vehicles:

```
// Abstract Class
abstract class Vehicle {
    abstract void startEngine();
    abstract void stopEngine();
    void honk() { System.out.println("Beep!"); }
}

// Interface
interface Flyable {
    void takeOff();
    void land();
    void fly();
}

interface Swimmable {
    void dive();
    void surface();
    void swim();
}

// Implementation
class Car extends Vehicle {
    @Override
    void startEngine() { ... }
    @Override
    void stopEngine() { ... }
}

class Airplane implements Flyable {
    @Override
    void takeOff() { ... }
    @Override
    void land() { ... }
    @Override
    void fly() { ... }
}

class SeaPlane implements Flyable, Swimmable {
    // ... implementations for Flyable methods
    // ... implementations for Swimmable methods
}
```

In this example, `SeaPlane` can fly and swim, demonstrating multiple inheritance of type. This is impossible with abstract classes in most languages.

Conclusion

Abstract classes provide a template and can enforce 100% abstraction for their subclasses. However, interfaces are still crucial for achieving flexibility, multiple inheritance, decoupling, and defining capabilities that can be added to unrelated classes. Abstraction alone isn't enough because it limits flexibility and creates tight coupling. Interfaces provide a more flexible and decoupled way to achieve abstraction by defining contracts that classes can implement regardless of their inheritance hierarchy.

(OOP) design principles.

1. Why We Need Interfaces When Abstract Classes Exist

While abstract classes offer a good degree of abstraction, we need interfaces primarily for two reasons: **Multiple Inheritance** and **Defining Contracts**.

A. Multiple Inheritance of Type (The Primary Reason)

Java (and many other OOP languages) prohibits a class from **extending** more than one class simultaneously. This prevents the "Diamond Problem" (ambiguity when a class inherits methods from two parent classes that share a common ancestor), leading to ambiguity about which ancestor's method implementation should be used.

- **Abstract Class Limit:** A class can only be a "child" of one parent class: `class Dog(Animal, Pet):` is illegal.
- **Interface Solution:** A class can **implement** any number of interfaces: `class Mallard(Swimmer, ZoneTrack, ZoneTrack2):` is legal.

This allows a single class to inherit the **type contract** (the shell) from multiple sources without inheriting conflicting implementations (the how). For example, a `Duck` class could implement both `Swimmer` and `ZoneTrack` interfaces, forcing it to define the specific behavior for both, but not lying to the implementation details of any one superclass.

B. Defining Contracts (Decoupling)

Interfaces are pure contracts that enforce a structure without providing implementation. This leads to **loose coupling**:

- Interfaces are typically used to define **API boundaries** between different modules or layers of an application.
- The code that uses the interface doesn't care about how the methods are implemented, only that they are implemented. This is crucial for **Dependency Injection** in frameworks like Spring, allowing the container to swap out different concrete implementations at runtime.

2. Why Interfaces Provide 100% Abstraction

An interface provides 100% abstraction because, historically (prior to Java 8's introduction of default methods), it contained **no implementation logic or state (fields) whatsoever**:

- **No State:** Interfaces cannot have instance variables; they can only contain `public static final` constants, which are shared across all implementations.
- **No Concrete Methods:** Before Java 8, all methods were implicitly `public abstract`. (They had a signature but no body.)

The result is total separation: The interface specifies the blueprint (the what), and the implementing class provides every single detail (the how). The user of the interface sees only the contract, achieving complete abstraction from implementation details.

Note on Default Methods (Java 8+): Since 8 introduced `default` and `static` methods in interfaces, which allows for some implementation, this technically means an interface can contain implementation. It was added primarily for backward compatibility (allowing new methods to be added to old interfaces without breaking existing implementations) and does not compromise the core goal of enforcing a contract.

3. Encapsulation vs. Abstraction

Encapsulation and Abstraction are two distinct, but complementary, pillars of OOP.

A. Encapsulation (Hiding the "How")

Encapsulation is the mechanism of **bundling data (fields) and the methods (procedures/etc) that operate on that data** into a single unit (a class) and **restricting direct access** to the internal state.

- **Goal:** To protect the integrity of an object's data and control how the data is modified.
- **Mechanism:** Using access modifiers like `private` for fields and providing controlled access via `public` getter and setter methods.
- **Focus: Data security and integrity.** It hides the internal details of how an object stores and manipulates its data.

Analogy: Encapsulation is like a **remote control**. You can change the channel using the public buttons (`setChannel()`), but you cannot directly touch the internal circuits (`private fields`).

B. Abstraction (Hiding the "Complexity")

Abstraction is the principle of **showing only essential information** and hiding complex implementation details from the user.

- **Goal:** To manage complexity by providing a simple, high-level view of an object or operation.
- **Mechanism:** Using abstract classes, interfaces, and abstract methods to define **what** an object does without specifying **how** it does it.
- **Focus: Complexity management.** It hides the implementation details of an operation.

Analogy: Abstraction is like driving a **car**. You use the steering wheel and pedals (the `doThisCommand()`) without needing to know the complex internal logic of the engine.

Ask Gemini

Tools

2.5 Flash

Gemini can make mistakes, so double-check it

Ask Gemini

Tools

2.5 Flash

Gemini can make mistakes, so double-check it

5 Whys and STAR Technique Expla...
Best Debrief Service Comparison
Settings and help